Title The Development of Test Action Bank for Active Robot Learning

Name Tao Cao

THE DEVELOPMENT OF TEST ACTION BANK FOR ACTIVE ROBOT LEARNING

By

Tao Cao

A thesis submitted to the University of Bedfordshire, in fulfilment of the requirements for the degree of Master of Science by research

November, 2009

ABSTRACT

In the rapidly expanding service robotics research area, interactions between robots and humans become increasingly common as more and more jobs will require cooperation between the robots and their human users. It is important to address cooperation between a robot and its user. ARL is a promising approach which facilitates a robot to develop high-order beliefs by actively performing test actions in order to obtain its user's intention from his responses to the actions. Test actions are crucial to ARL.

This study carried out primary research on developing a Test Action Bank (TAB) to provide test actions for ARL. In this study, a verb-based task classifier was developed to extract tasks from user's commands. Taught tasks and their corresponding test actions were proposed and stored in database to establish the TAB. A backward test actions retrieval method was used to locate a task in a task tree and retrieve its test actions from TAB. A simulation environment was set up with a service robot model and a user model to test TAB and demonstrate some test actions.

Simulations were also performed in this study, the simulation results proved TAB can successfully provide test actions according to different tasks and the proposed service robot model can demonstrate test actions.

Keywords: service robots, human-robot interaction, robot active learning, classification, test actions

DECLARATION

I declare that this thesis is my own unaided work. It is being submitted for the degree of Master of Science by Research at the University of Bedfordshire.

It has not been submitted before for any degree or examination in any other university.

Name of candidate: Tao Cao

Signature: Tao Cao

Date: 11/02/2010

ACKNOWLEDGMENT

First and foremost, I am grateful to my supervisor Dr Dayou Li of the Department of Computer Science and Technology, University of Bedfordshire. It is due to his excellent academic guidance and tireless support that this research work and this dissertation have been possible. He has been, and continues to be, a wonderful and patient supervisor.

Thanks to Xiao Guo, David Fletcher and Kenz A.Bozed in IRAC, for their help with experiments and offering valuable suggestions.

Lastly, I would like to thank my family, for their unwavering support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

1.1 Motivation

Service robots include home or personal service robots, entertainment robots, education robots, medical robots, healthcare and rehabilitation robots and rescue robots. They are expected to provide services to their human users at home and within the workplace. For example, they will be able to assist aging population in terms of living in their own homes, to assist health workers perform routine procedures, to increase the effectiveness of surgical procedures in hospitals, etc.

In the rapidly expanding service robotics research area, interactions between robots and humans become increasingly common as more and more jobs will require cooperation between robots and their human users. It is important to address cooperation between a robot and its user. When working with human users, the robots inevitably need to share the human environment and to participate in joint activities with the users. That is, human-robot cooperation is needed. Since human environments are complex, dynamic, uncontrolled, and difficult to perceive reliably, to achieve human-robot cooperation, the robots are required to understand humans' intention and preferences. Based on their understanding, the robots can coordinate and adjust their behaviours to provide desired assistance and services to the users as capable partners. However, a robots' understanding of their users' intention is still an exceptionally difficult challenge.

At present, there are existing approaches to pursue research in humans and robot cooperation and the development of the robots' understanding of humans' intention. Breazeal et al (2004) insist that in human-robot cooperation, a robot requires an understanding of its uses' intentions and desires in order to behave as a partner rather than just a tool. They demonstrated a humanoid robot that works in cooperation with a human via social cues such as pointing, gazing and natural language. Oliver etc. (2005) presented an approach to human-robot cooperation that allows a robot to actively take further actions to confirm human's intention in cases where the robot cannot recognise the intention according to the human's current behaviour. Calinon and Billard (2006) used imitation based learning to enable the robot to recognize the user's intention when it captures the same gestures. Tapus and Mataric (2007) proposed a reinforcement learning based approach to robot behaviour adaptation, the aim of this approach is to develop a robotic system capable of adapting its behaviours according to the user's personality, preference, and profile in order to provide an engaging and motivating customised protocol. Inspired by discovery learning theory which encourage learners to acquire information by performing their own experiments; Li et al (2008) proposed active robot learning (ARL).

ARL does not rely on social cues and explicitly defined award functions. ARL is an active learning approach. This means that the robot decides when to learn and what to learn. In ARL, a robot actively performs test actions in order to obtain its user's intention from their responses to the actions. Test actions are crucial to this approach. First, test actions should naturally represent mappings from the user's intention to his responses with respect to the actions. This kind of mapping will enable the robot to recognise the user's intention according to his responses. Second, these actions should

be classified and organised in a hierarchical structure according to corresponding tasks. This is because the robot can perform numerous similar tasks, the number of test action for all tasks can be huge while the test actions for the similar tasks can be the same. Poorly organised test actions will affect the efficiency and effectiveness of ARL process.

1.2 Aim and Objectives

This study aims at the development of a test action bank (TAB) for ARL, including establishing a verb based task classifier which is used to classify a user's commands into task categories, choose suitable test actions from TAB according to the chosen task category and developing the TAB itself. This study will also set up a 3-D virtual reality environment to test the applicability of TAB and the task classifier.

The main objectives are as follows:

- To develop scenarios and typical tasks for choosing test actions used to establish TAB
- To identify and define the features of typical tasks, all the taught tasks will be classified into categories and organized in a hierarchical structure to support task classification.
- To define the verb's features using relevant nouns and develop a task classifier based on these feature vectors to classify user's commands.

- To store taught tasks in a relational database in a hierarchical structure, with the top are very abstract tasks and the very specific ones are arranged at the bottom.
- To establish TAB using relational database and choose suitable test actions from TAB according to corresponding task.
- To develop a 3-D virtual reality environment with a robot and a user model in order to simulate and test the applicability of the TAB and the task classifier.
- To integrate the task classifier and TAB with the simulation environment.
- To test the TAB and task classifier using typical tasks and scenarios.

1.3 Scope and methodology of this study

This study focuses on the development of TAB database and how to choose suitable test actions according to user's commands and different tasks. This study assumes the service robot used is capable of converting user's commands, which may not in text format into text so that tasks described by the text commands can be classified and test actions can be chosen accordingly.

The test actions are proposed to consider taught tasks only, while untaught tasks will not be considered here because of safety and other social issues. Four typical taught tasks were discussed in this study, as these are sufficient for explaining how the TAB is established and how it works.

Tasks can be classified into categories and a task classifier is proposed to choose test actions from TAB for each task. As the task classifier relies on verbs of a command to carry out classification, some commands without a meaningful verb will be treated as invalid commands and the classifier will ask the user to provide another command. The use of verbs to classify commands, on the other hand, limits the application of this task classifier, but this shortcoming can be overcame by implying more natural language processing techniques in future work.

ARL emphasises the robot's capability of intention recognition via active (not passive) learning. During the ARL process, a robot can perform numerous tasks and many of them are similar and can use the same test actions. Therefore, test actions are organized in TAB in a hierarchical structure according to typical taught tasks to support task classification. Taught tasks are represented from abstract level to specific level, that is, the top of this hierarchy contains very abstract tasks and the very specific ones are arranged at the bottom. For example, "pass an object" is a highly abstract task, and therefore is arranged on the top. Whilst, "pass a spoon", "pass a drink" and "pass a fruit" are less abstract and because of this they are arranged one level lower than "pass an object". "Pass a tea spoon" and "pass a table spoon" are more specific so they are arranged at the bottom. With this hierarchy, test actions related to the tasks of four categories, together with their features, are stored into different tables of a relational database. A given task can will then be classified to see which category it belongs to and then test actions of the typical taught task from corresponding categories will be used as the test actions in order to recognise a human's intention which is needed to fulfil the task. Using this strategy, the number of test actions for numerous tasks can be limited.

1.4 Structure of Dissertation

Chapter 2 provides a survey on how to enable service robots to understand their user's intentions. This chapter describes and investigates five approaches in the research area of intention recognition and human-robot cooperation. These approaches employ various methods, such as verbal communication, social cues, purposely defined uncertainty measure/award function and proactive test actions.

Chapter 3 presents the development of a verb-based task classifier, which were used to process user's commands and find out its corresponding task category. The classifier uses the API from the link grammar parsing system to characterize the command, such as syntactical parsing, semantic role annotation and PoS tagging. This initial processing can identify the verb and other PoS of the commands, with ".n" for nouns, ".v" for verbs, ".a" for adjectives and ".e" for adverb etc. All of these tagged words will be stored as clues for classifying this command. The verb will be used to find the command's category and to find relative test actions from TAB needed to retrieve the entire clues according to the order of the adverb, adjective, noun, and verbs until the suitable test actions are found. Our methodology can make sure the valid user command will retrieve its test actions.

Chapter 4 gives details of the establishment of the TAB database and the retrieval process of relative test actions according to user commands and task categories. Four typical tasks and their subtasks are displayed and stored in a MySQL database in a hierarchical structure to enable task classification and test action retrieval. The TAB

database is composed of three types of tables, namely "task category", "task tree structure" and "test actions". Four typical tasks are stored in the task category table; there are four task tree structure tables to implement the hierarchical structure of four typical tasks and their subtasks correspondingly. All the test actions are stocked in the test action tables. After the task classifier indicates the verb of a command, one of these verbs can be used to find out this command's category from the task category table, as this category is a very abstract one, more specific tasks will be retrieved from a special task tree structure table. The test action retrieval process will continue until the task is precise enough and its test actions can be retrieved from a test actions table.

Chapter 5 introduces the implementation of 3-D virtual reality environment and the model of a service robot and its user, which were used in this study for the purpose of simulation and evaluation of the applicability of the TAB and the task classifier. All of the service robot, the user and the 3-D environment were created using open dynamic engine (ODE) toolkit, which is an open source, high performance library for simulating rigid body dynamics and has advanced joint types and integrated collision detection with friction. Both the robot and the user model consist of a base and an arm, and can move around and mimic the humans' arm movement

Chapter 6 shows the integration and test of task classifier and TAB with the simulation environment and simulation results. After the integration the service robot model can perform corresponding test actions according to user's commands. That is, the task classifier accepts and processes user's command to find out its category and corresponding test actions from TAB. The robot model then will perform each of these test actions to display the result.

Chapter 7 gives conclusions and further work.

CHAPTER 2. LITERATURE REVIEW

In order to enable a robot to cooperate better with its user, the robot needs to be able to predict what the user will do next. The prediction can be made based on the users' intention and/or preference, that is, the high-order beliefs of the robot. This chapter introduces service robot and some research approaches aiming at the development of such beliefs for robots.

2.1 Service robotics

According to the IFR (International Federation of Robotics), a service robot is a robot which operates semi or fully autonomously to perform services useful to the well being of humans and equipment, excluding manufacturing operations (SRIC-BI, 2008). Service robots include home or personal service robots, entertainment robots, education robots, medical robots, healthcare and rehabilitation robots and rescue robots. They are expected to provide services to their human users at home and within the workplace. For example, they will be able to assist aging population in terms of living in their own homes, to assist health workers perform routine procedures, to increase the effectiveness of surgical procedures in hospitals, etc.

Service robotics is an area of increasing interest and investment, especially in many countries that are facing the challenge of an aging society, such as US, UK and Japan. In the future society, with the development of service robotics, service robot has potential to help the elderly people to live independently for longer. The robot could

cook or fetch meals for the elderly, clean their rooms, toilets and even handle tasks such as bathing, dressing or supporting users walking, sitting down or standing up. Equipped with special sensors a service robot can monitor users' health condition on a regular basis, it can take blood pressure, measure body temperature and heartbeat rate, this function is very useful for healthcare service robots.

Service robot can provide both the elderly and the rest of us with a great deal of service. It can also be used in hospital to help carry medicine, blood samples, assist surgeons in surgeries; in areas such as household assistance and tasks dangerous for humans, like fire fighting and bomb-disposal; in space applications where human astronauts and robot need to collaboratively assemble parts.

In the rapidly expanding service robotics research area, how to make the robot adaptable to new tasks and environments, how to enable robots capable to interact with humans become increasingly common, as more and more jobs will require cooperation between robots and their human users. It is important to address cooperation between a robot and its user. When working with human users, the robots inevitably need to share the human environment and to participate in joint activities with the users. That is, human-robot cooperation is needed. Since human environments are complex, dynamic, uncontrolled, and difficult to perceive reliably, to achieve human-robot cooperation, the robots are required to understand humans' intention and preferences. Based on their understanding, the robots can coordinate and adjust their behaviours to provide desired assistance and services to the users as capable partners. However, a robots' understanding of their users' intention is still an exceptionally difficult challenge.

2.2 Intention recognition based on dialog and gestures

Breazeal et al (2004) insist that in human-robot cooperation, a robot must be socially intelligent and must understand its use's intentions and desires in social terms in order to behave as a partner rather than just a tool. They demonstrated a humanoid robot (called Leo) to work cooperatively with a human on joint tasks.

Humanoid robot Leo has the capabilities of speech recognition and understanding, and vision. It also has simple manipulation skills. The robot can also communicate with its user during collaboration using a variety of gestures and other social cues. For example, a quick nod means understanding the user's words, a confirming nod indicates understanding the utterance, and displaying a look of confusion indicates having problems in understanding the speaker. A list of social cues used in their work is given in Table 1.

Cooperation between Leo and its uses was demonstrated using the following scenarios:

- Button-one task (toggling a single button): when the user said "let's do task Button-one", the robot looked at the button to acknowledge he understands the task and then pointed to himself to show he can have a go first. When the user said "ok, you go", the robot then pressed the button and looked at the user to signal the end of his turn and nodded shortly to indicate the end of the task.

Table 1.1: Summary of the humanoid robot Leo's social cues:

Social Cue	Communicated Intention	Interaction Function
Follows gesture to Object of Attention (OOA)	Establish OOA common ground	OOA set & ready for labeling
Point to object, look to object	Identify a particular object as referential focus (e.g., demonstrate correct association of name with object).	Confirm mutual belief about a particular object referent (e.g., successful identification of the target)
Confirming Nod (short)	Confirmation (e.g., OK, got it)	Update common ground of task state (e.g., attach label, start learning, etc.)
Affirming Nod (long)	Affirm query (e.g., Yes, I can)	Affirmation to query
Leaning forward and raising one ear towards human	Cannot understand (unable to recognise/parse speech)	Cues the human to repeat what was last said
Cocking head and shrugging (express confusion)	Cannot perform the request (lack of understanding)	Cues the human to add information or rectify shared beliefs (request clarification or elaboration)
Shake head	Cannot perform the request (lack of ability)	Cues that robot is not able to perform the request
Small ear perk and slight lean forward	Attention to human voice	Cues that robot is listening and attending to human
Break gaze, perform action	Acquire floor and begin turn	While the robot looks away, its turn is in progress
Looks back at human, arms relaxed	Turn is completed	Relinquish turn back to human
Looks back at the human, points to himself.	Gaze shift used to set turn taking boundaries. Gesture indicates perceived ability to perform an action.	Self-assessment and negotiating Sub-plan meshing.
Glances to the OOA, and opens arms to the human.	Detects inability to perform needed action on OOA, asking for help.	Request human partner completes the step.
Looks at workspace.	Checks and updates change in task state due to own or other's act.	Acknowledge change in task state to other.
Eyes follow human action.	Acknowledges partner's action, maintains common ground.	Acknowledge that action is Completed by other agent.

- Button-one-and-two task: when the user said "let's do task Button-one-and-two", the robot looked at the buttons to acknowledge he understands the task and pointed to himself to show that he can have a go first. When the user said "ok, you go", the robot then pressed one of the buttons and looked at the second button. When the user said "ok, you go" again, the robot pressed the second button and looked at the user to signal the end of his turn and nodded shortly to indicate the end of the task.

In these scenarios, the user expressed his intention verbally. The robot recognised the user's intentions by performing speech recognition and understanding. The robot also expressed its own intentions using gesture and social cures.

2.3 Proactive human-robot cooperation

Oliver etc. (2005) presented an concept of proactive execution of robot tasks in the context of human-robot cooperation with uncertain knowledge of the human's intentions, this approach allows a robot to actively take further actions to confirm human's intention in cases a intention cannot be clearly recognised according to the human's current behaviours. The structure of the robot system they proposed is depicted in Fig. 1. The two key modules are the Intention Recognition that determines the human user's intentions and the Planner that executes the appropriate tasks based on those intentions. The Sensors and Actuators form the interface to the outside world. The Motion Control module controls the motion of the robot to complete a given task

and the Database contains rules for intention recognition and also for planning further actions.

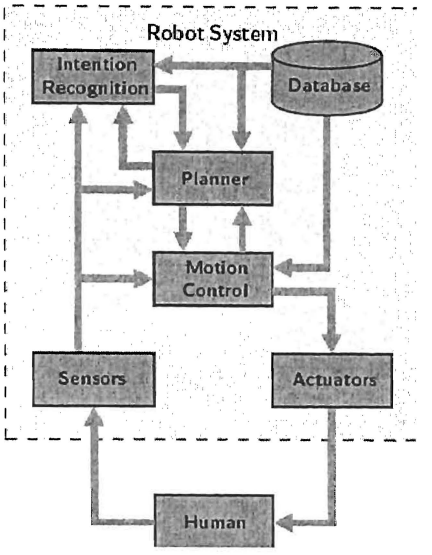


Fig. 1 Robot system structure for proactive human-robot cooperation

The core of the intention recognition module is a Hybrid Dynamic Bayesian Networks (HDBN) which makes decisions on human's intention using probabilistic methods according to his behaviours (obtained through the sensors) and rules provided by the Database. The HDBN will extract several hypotheses of the path that the human will move along in the near future and the type of interaction he desires to have with the robot, that is, the possible tasks that the human is likely to complete in the near future. Thus the Intention Recognition module makes an effort to understand as much of the nonverbal communication as possible (provided by the human), and the result is fed to the Planner module. In case the information about the human intention is too uncertain the Planner is forced to execute tasks proactively, pick an intention and pretend that this is the desired intention, and select an appropriate task. Subsequently, the system will start executing this task, closely monitoring how the values from the

intention recognition develop. To deal with the case when there are similar probabilities tip in favour of the chosen intention, and keep executing the task; on the other hand, if it becomes clear that this task does not match the human's intention, stop execution. The challenge here is the optimal selection of an intention from the two or three candidates. In cases where no intention was recognized with a sufficient certainty, the Planner will select either an idle task or a task that tries to capture the human user's attention and communicate that the robot is idling and waiting for a command.

2.4 Learn by imitation

Robot learning plays an important role in background knowledge building, motivation establishment and preference identification. The current robot learning approaches include imitation learning. The imitation based learning uses social cues such as pointing and gazing to indicate what the user intended to do next (Dillmann 2004, Breazeal et al. 2005, Calinon and Billard 2006). The user first teaches a robot by demonstrating gestures, for example, pointing to and gazing at an object. These gestures serve as social cues of his interest on the object. Then the robot imitates the gestures for the user's approval. This imitation process enables the robot to recognise the user's intention when it captures the same gestures.

Experiments carried out in Calinon and Billard (2006) are described below: During a first phase of the interaction, the designer demonstrates a gesture in front of a robot. The robot then observes the designer's gesture. Joint angle trajectories are collected

from a motion sensor. The second phase begins when the robot collected the different movements of a user. The robot compares the gesture it collected with the gesture stored earlier and finds the necessary cues. Then the robot points at an object that the user is likely to be interested in. The robot then turns to the user for evolution of its selection. The designer signals to the robot whether the same object has been selected by nodding/shaking his/her head.

In imitation learning, a Hidden Markov Model (HMM) with a full covariance matrix is used to extract the characteristics of different gestures which are used later to recognise gestures from the user. The characteristic of a gesture is expressed by transition across the state of the HMM. Using such a model requires the estimation of a large set of parameters. An Expectation-Maximisation (EM) algorithm is used to estimate the HMM parameters. The estimation starts from initial estimates and converges to a local maximum of a likelihood function. It does this by first performing a rough clustering and then carrying out an estimate a Gaussian Mixture Model (GMM). Finally, the transitions across the states are encoded in a HMM created with the GMM state distribution.

2.5 Reinforcement learning system

Tapus and Mataric (2007) proposed a reinforcement learning based approach to robot behaviour adaptation. The aim of this approach is to develop a robotic system capable of adapting its behaviours according to the user's personality, preference, and profile in order to provide an engaging and motivating customised protocol.

In this learning approach, a robot incrementally adapts its behaviour and its expressed personality as a function of the user's extroversion-introversion level and the amount of performed exercises. Then the robot attempts to maximize that function.

The learning process consists of the following steps:

- Parameterisation of the behaviour
- Approximation of the gradient of the reward function in the parameter space
- Movement towards a local optimum.

The main goal of this robot behaviour adaptation system is to optimise three main parameters (interaction distance, speed, and verbal cues) that define the behaviour of a robot, so that the robot can adapt itself to the user's personality and improve its task performance. Task performance is measured as the number of exercises performed in a given period of time. The learning system changes the robot's personality which is expressed through the robot's behaviour to maximise the task performance.

2.6 Active robot learning (ARL)

Inspired by discovery learning theory which encourages learners to acquire information by performing their own experiments; Li et al (2008) proposed active robot learning (ARL).

2.6.1 Active learning process

ARL is inspired by discovery learning, which takes place in problem solving situations where the learner draws on his own experience and prior knowledge. Discovery learning can be simply described as "learn by doing". It has then been developed into a method of instruction through which learners interact with their environment by exploring and manipulating objects, wrestling with questions and controversies, or performing experiments.

The property of "learning by doing" makes it suitable for the robots to develop high-order beliefs. Belief is the psychological state in which an individual (including cognitive robots) holds a proposition to be true. In computer science, the decision on whether the proposition is true (uncertainty) can be made by looking at the evidence of other related propositions (Dempster 1968, Shafer 1976). The collection of relevant evidence is, therefore, an important step in the process of building up beliefs. In situations where service robots co-work with their users, to build up their high-order beliefs of the users' intention and preference, the robots also need to collect evidence which may not be seen at first glance.

ARL allows a robot to actively set hypotheses about its human user's intentions and perform guided tests on the users to predict their hidden intentions. The users' reactions to the tests are considered as evidences that will either confirm or reject the hypotheses. ARL emphasises the robot's capability of intention recognition via active (not passive) learning. With discovery learning capability, the robots will be able to perform experiments when they are not sure what their human counterparts intend and

prefer to do. By doing this, the robots can discover evidence which is required but not seen at first glance to build up high-order beliefs.

During ARL process, when a robot is working together with its user to fulfil a cooperative task, test actions are crucial for guaranteeing such task can be successfully done. Test actions should naturally represent mappings from the user's intention to his responses with respect to the actions. This kind of mapping will enable the robot to recognise the user's intention according to his responses. For example, in the situation where a robot helps its user to lift an object from the ground and the robot realises that the user stopped at certain height, the robot will need to find out whether the user intends to lift the object only to that height or to return the object down to the ground because of changing circumstances. The robot can test the user by gradually lowering the object and seeing how the user will respond. If the robot perceives the same action from the user, it can then regard the response as the evidence of changing of mind. If the robot perceives no action from the user, it can view this response as evidence of the preference of keeping the object at that current height.

Because a robot can perform numerous similar tasks, the number of test action for all tasks can be huge while the test actions for the similar tasks can be the same. So test actions for ARL should be well organised to improve the efficiency and effectiveness of ARL process. As different tasks will require different actions and test actions. A group of similar tasks can share the same sets of test actions. Therefore, tasks will be classified into categories at an abstract level and test actions will organised in a

hierarchical structure according to corresponding tasks in order to avoid duplicate test actions in test action bank.

The discovery learning method has been also employed in supervised machine learning, and is known as active machine learning (AML), as a resolution to the problem of lacking expensive labelled training examples. AML has also been used in robot control to model the inverse dynamics of a robot arm with high model uncertainty (Robbel and Vijayakunar, 2007).

ARL differs from AML because ARL requires a robot to carry out experiments to generate data (evidence), whilst AML only searches for and evaluates available data.

2.6.2 Structure of ARL

The overall structure of an ARL system is shown in Figure 2.2. The system consists of an action bank which stores actions that can be taken to test its users, an inference engine which reasons about what actions are to be taken for a specific purpose, a moment determination mechanism to decide the moment of test, an intention identification mechanism to interpret responses of the users and to identify intention and preference, and an intention model which represents intentions.

Test actions are those that can be taken to test the users. They are associated with conditions and stored in the action bank. Each test action stored in the action bank has a name and content which is the kinematics of the robot. The conditions express reasons for performing the actions and are represented as propositions. For example,

if a robot hands over a glass of water to its user, it would need to check whether the user intends and is ready to take over the glass. One of the test actions for testing the user in this case is to slightly loosen the glass and the condition associated is to confirm the user's intention of taking over the glass. The actions and the associated conditions can be designed by robot designers before the robots are deployed.

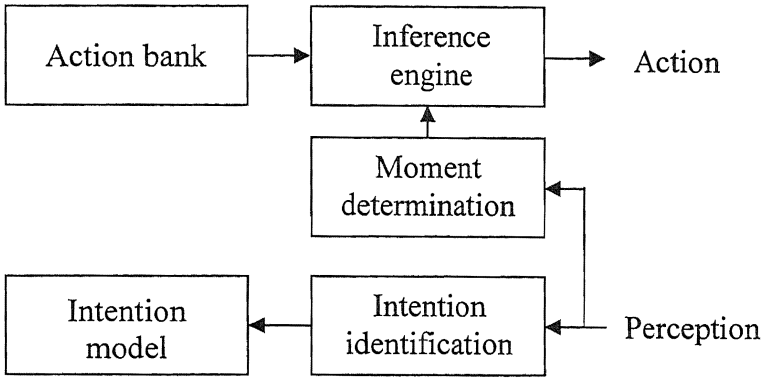


Figure 2.2 Structure of ARL System

The inference engine selects a test action from the action bank to conduct a specific test. As the actions are associated with conditions in the action bank and the associations actually represent causal relations (implications) from the conditions to the actions, the selection of an action can be carried out using a standard forward reasoning approach.

The moment determination mechanism decides the starting time for testing the user and triggers the action bank to send out a test action. There are, in general, two moments where a robot needs to test the user for intentions. The first is the moment

before the last action in the course of the completion of a task. Taking the example of getting a drink for the user, before a robot finally takes the action of releasing a glass, it needs to find out whether the user intends and is ready to take over the glass. In the example of assisting a human getting up from a chair, the robot has to make sure that the user intends and is ready to stand alone before releasing his arm/hand. The second is the moment when a robot feels its user stops doing what he originally intended to do. The robot will need to find out whether the user changed his mind or not for further cooperating with him. In this situation, the robot will rely on its perception to detect this change.

CHAPTER 3. TASK CLASSIFIER BASED ON VERB'S FEATURE VECTOR

Different tasks will require different actions and test actions. A group of similar tasks can share the same sets of test actions, on the other hand. Therefore, it is necessary to classify tasks into categories at an abstract level.

A service robot can fulfil tasks according to the user's commands; most of these commands are declarative sentences containing verbs. These verbs, which can be identified using nature language processing techniques, can be used to extract tasks from commands and classify these tasks into typical task categories. This idea will be used in this study to extract tasks from user's commands and to classify them. Four typical tasks are generalized as the most useful and worthwhile tasks from the common service tasks of service robots, each task has a verb with a feature vector as its keyword to present this task's feature. The identified verb from a command also has its own feature vector, which will be compared with keywords' feature vectors to find out to which category this command belongs. A task classifier was developed to carry out the comparison.

3.1 Typical tasks

Service robots are likely to become an important part of human society. The deployment of robot servants at home will make a significant difference to people's daily life, especially for elderly and disabled people.

The typical tasks mentioned here have the following meanings in the service robot area: these tasks involve cooperation between robots and their users; there are great expectations for various applications of these tasks; more importantly, these tasks have high practical values in terms of improving service robots' capabilities. In summary these tasks are useful and warrant further study.

3.1.1 Task: pass an object

Transferring of objects between robots and humans is a fundamental way to coordinate activity and cooperatively perform useful work. Consider, for example, robots working alongside people in their homes and workplaces, such as helping an elderly person living at home, helping with household chores, etc. When the user needs something that is not near him/her, he/she can ask a robot to get it. After successfully finding the requested object, the robot can take the object to the user and hand it to him/her. This object can be a glass of water, a newspaper, an apple or a kind of food. During this kind of service, successfully transferring the requested object from the robot to the user is critical, we define this task with the verb "pass" and these kind tasks as "pass an object". Passing on object is common and frequently occurs in our daily life.

To naturally and intuitively accomplish the "pass an object" task between a robot and its user, the robot must understand its user's intention, which means the robot should clearly know whether the user is ready to take the delivered object. If the robot releases the object before its user is ready to hold it, the object will fall to the floor; on

the contrary, if the user is ready to take the object but the robot doesn't release it, the cooperation between them will fail.

3.1.2 Task: support by arm

The ability to walk is one of the most basic needs of a human being, however, many people, for various reasons, cannot walk unaided. For example, people who are suffered stroke with one side of their body paralysed, people who are affected by arthritis or other chronic diseases and some older people who are infirm. Walking is especially important for post-operative recovery patients who, on the one hand, are too weak to walk independently and, on the other hand, have a walk everyday in order to promote the body's recovery. The current solution is that someone, such as a family member, a friend or a healthcare assistant, helps them to walk. However, the problem is that it is difficult to provide care 24 hours a day, 7 days a week. In an ideal situation a robot may replace human helpers to do this kind of work. This will bring great convenience to the users. For example, they can even ask the robot assist them to go out shopping.

The process of a robot assisting its user to walk should be user-centred, which means the robot should help its user according to his goals and desires. Therefore, the robot needs to understand its user's intention. Consider a robot helping its user walking at home as an example. The robot is expected to know when and where the user needs help – he/she may want to walk around the living room, to go to upstairs/downstairs, or to sit down for a rest.

All of these situations are likely to happen; this requires the robot to act promptly and precisely inferring its user's intentions. Only once the robot has such a capability, can it appropriately adjust its behaviour to help the user.

3.1.3 Task: feed the user

Eating food is indispensable and absolutely necessary to human life. Some people can not eat by themselves, they need someone help them to eat, such as some sick people, elderly people, or post-operative patients, these people are too weak to eat on their own.

Enabling a robot to assist its user to eat is still a big challenge within the robotics area, because the capable robot needs to recognise when and how much a user will eat, in addition to what the user wants to eat. The most significant aspect is that the robot can understand whether the user is full or not.

3.1.4 Task: help a user to move an object

Moving an object together with its user is another typical task for a service robot. This kind of task is useful when the user wants to move heavy furniture like beds or a fridge. Firstly, the user must let the robot know where to relocate the object. After that they will begin to move it, during the moving process, there can be many situations and possibilities, for instance, the user maybe move it up, down, left, right, forward, backward or just stay still for a moment. Any of the above actions are possible, and the robot needs to actively respond to any kind movements. For example, in the

circumstance where a robot helps its user to lift an object from the ground and the robot realises that the user stopped at certain height, the robot will need to find out whether the user decides to lift the object only to the current height or to put the object down to the ground because he changes his mind. The robot can test the user by slightly putting down the object and see how the user response. If the robot perceives the same action from the user, it can then regard the response as the evidence of changing mind. If the robot perceives no action from the user, it can view this response as evidence of the preference of keeping the object in current position.

3.2 Verb's feature vector

Verbs of robots' user commands can represent actions, which can be used to match actions of tasks. Some verbs will be used as keywords for user's command. Our approach is using these keywords to identify and to classify tasks.

3.2.1 Word's feature

As mentioned above, "pass an object" and "help user move an object" are two typical tasks. The commands for these tasks can be: "please give me a cup of tea", "bring a glass of water to me", "help me take this table", or "please help me carry this TV", etc. In order to pick up a verb from a user's command, natural language processing (NLP) techniques are used to process the commands, which are considered as text input. The initial input consists of a target word along with a portion of the text in which it is

embedded, which is called its context. Here, the target word is the verb and the sentence is its context.

After initial processing, the input is then reduced to a fixed set of features that capture information relevant to identifying the meaning of the target word, then the relevant features will be selected to encode in a usable form. Usually, a simple feature vector is composed of numeric or nominal values. Collocation feature is one of the most used features (Jurafsky, Martin; 2000a). In general collocation refers to a quantifiable position-specific relationship between two lexical items. Collocation features encode information about the lexical inhabitants of specific positions located to the left and right of the target word. Typical items in this category include the word, the root of the word, and the word's part-of-speech (PoS, e.g. noun, verb, adverb, and adjective) (Jurafsky, Martin; 2000b). This type of feature is effective at encoding local lexical and grammatical information that can often accurately isolate a given sense. In this study, the words themselves (or their root) serve as features. The value of the feature is the number of times the word occurs in a region surrounding the target word. This region is most often defined as a fixed size window with the target word at the centre. To make this approach manageable, a small number of frequently used content words are selected for use as features. This kind of feature is effective at capturing the general topic of discourse in which the target word has occurred. This, in turn, tends to identify senses of a word that are specific to certain domains.

3.2.2 Defining a verb's feature vector using relative nouns

In terms of human being's experience on the use of language, some verbs have different meaning; however, they can have the same meaning when they collocate with some relevant nouns (Guo. X, 2009). A simple example is "give" has not been defined as a synonym of "pass" in the electronic dictionary such as WordNet. However, if user says "Give me a cup of tea" or "pass me a cup of tea", obviously "give" is a synonym of "pass" in this command (sentence). Therefore, we can find the meaning of the target words through the analysis of the structure of a sentence. One approach is to tag PoS to each word in the sentence first. The syntactical structure of "give me the drink" is "verb+pronoun+determiner+noun" which is the same as "pass me the drink". Secondly, remove pronouns and determiners which are considered not important to the meaning of a sentence. Henceforth, "give+drink" as collocation of words is remained. The collocation of those words is the same as "pass+drink", and we can therefore conclude that "give" is a synonym of "pass". We can then record "give" as a synonym of "pass" in our own dictionary. This is a simple approach to discriminate whether a word is a synonym of a target word. In following sections, a method of tagging PoS of words in a sentence will be introduced.

As mentioned above, when "pass" and "give" collocate with "drink", in this collocation, the two verbs have the same meaning, but usually they do not. This means these two words have similar collocation features, the collocation feature of verbs will be used to classify the verbs.

Based on the concept of collocation feature, we propose a method to define a verb's feature vector using relevant nouns. Take the verb "move" and "pass" for example, when using "move", we can say "move the table", "move the bed" or "move the box",

but not "pass the table" or "pass the bed", correspondingly, "move me a cup of tea" or "move me a glass of water" does not make any sense. Instead, we would say "pass me a cup of tea", "give me a glass of water". So we choose some nouns to compose a vector to determinate the target verbs' feature vectors, and using these feature vectors to classify these verbs and their relative commands into task categories. The task categories are chosen according to the four typical tasks. The corresponding nouns we select to define a verb's feature vector are: cup, beer, tools, bed, fridge, box, soup, water, apple, stand, sit, and walk, using them to constitute a vector as shown below:

{Cup, beer, tools, bed, fridge, box, soup, water, apple, stand, sit, and walk} (3.1)

This vector is used to define a chosen verbs' feature vector. If a verb can collocate with the relevant noun, the corresponding noun will be replaced by "1", if not, replaced by "0". Using such a method, the corresponding feature vectors for 'move' and 'pass' are {0,0,1,1,1,1,0,0,0,0,0,0} and {1,1,1,0,0,1,1,1,1,0,0,0}. These feature vectors are a word's collocation feature, so verbs with similar collocation features can be defined using the same feature vector.

The chosen verbs for commands used for the four typical tasks are: "pass", "give", "bring", "move", "take", "carry", "feed", "need", "drink", "support", "stand", "sit" and "walk". These verbs are divided into four categories, with "pass", "move", "feed" and "support" as each task category's keyword. The feature vectors of these verbs are shown in table 3.1.

Table 3.1 The verbs' feature vectors

noun / verb	Cup	beer	tool	bed	fridge	box	soup	water	apple	stand	sit	walk
pass	1	1	1	0	0	1	1	1	1	0	0	0
give	1	1	1	0	0	1	1	1	1	0	0	0
bring	1	1	1	0	0	1	1	1	1	0	0	0
move	0	0	1	1	1	1	0	0	0	0	0	0
take	0	0	1	1	1	1	0	0	0	0	0	0
carry	0	0	1	1	1	1	0	0	0	0	0	0
feed	0	1	0	0	0	0	1	1	1	0	0	0
need	0	1	0	0	0	0	1	1	1	0	0	0
drink	0	1	0	0	0	0	1	1	1	0	0	0
support	0	0	0	0	0	0	0	0	0	1	1	1
stand	0	0	0	0	0	0	0	0	0	1	1	1
sit	0	0	0	0	0	0	0	0	0	1	1	1

3.2.3 Distance between verbs

After defining a verb's feature vector, we consider some of these verbs have the same meaning and represent the same task. These verbs are called the synonyms. The synonyms selection depends on two aspects in NLP. One aspect is related to some synonyms of a word are usually defined on the sense inventory. Those synonyms are collected in terms of human being's experience on the use of language. The other aspect is some synonyms of a word have not been defined on the sense inventory. Therefore, those synonyms should be explored by using contextual information. As the verbs' feature vectors have been defined, we need to use these feature vectors to discriminate between verbs.

Latent Semantic Analysis (LSA) is a corpus-based measure of semantic similarity (Foltz, 1998). It provides a way of overcoming some of the drawbacks of standard vector space model. In fact, LSA similarity is computed in a lower dimensional space,

in which second-order relations among terms and texts are exploited. We choose standard cosine similarity method to calculate the distance between the feature vectors. Cosine similarity is a measure of similarity between two vectors of N dimensions by finding the angle between them, often used to compare documents in text mining (S. Deerwester, 1990). Given two vectors of attributes, $A=\{x_1,x_2,\ldots x_n\}$ and $B=\{y_1,y_2,\ldots y_n\}$, the cosine similarity, θ, is represented using a dot product and magnitude as follow:

$$similarity = \cos ine(\theta) = \frac{A \bullet B}{\| A \| * \| B \|} = \frac{\sum_{i=1}^{n} x_i * y_i}{\sqrt{\sum_{i=1}^{n} x_i^{\,2} \sum_{i=1}^{n} y_i^{\,2}}} \tag{3.2}$$

The result of cosine (θ) is in the range of [-1, 1]. A value of -1 means the meanings that A and B are exactly opposite, 0 means the meanings are independent, and 1 means the meanings are exactly the same. Values in between indicate intermediate similarities or dissimilarities. The following section discusses how the task classifier finds the target verb and its feature vector of a command and calculates the distance between the target verb and the four keywords.

3.3 Task classifier

The task classifier uses a Link Grammar Parser to identify the verb of a command, and then get the verb's feature vector and use it to find out the command's category.

3.3.1 The Link Grammar Parser and its API

A Link Grammar Parser (LGP) is a syntactic parser of English, based on link grammar, an original theory of English syntax (Daniel Sleator and Davy Temperley, 1991). Given a sentence, the system assigns to it a syntactic structure, which consists of a set of labelled links connecting pairs of words. The parser also produces a "constituent" representation of a sentence (showing noun phrases, verb phrases, etc.).

The parser has a dictionary of about 60000 words. It has coverage of a wide variety of syntactic constructions, including many rare and idiomatic ones. The parser is robust; it is able to skip over portions of the sentence that it cannot understand, and assign some structure to the rest of the sentence. It is able to handle unknown vocabulary, and make intelligent guesses from context and spelling about the syntactic categories of unknown words. It has knowledge of capitalisation, numerical expressions, and a variety of punctuation symbols.

The entire system is available for download on the Internet. The system is written in generic C code, and runs on any platform with a C compiler. There is an application program interface (API) to make it easy to incorporate the parser into other applications. The API is written in ANSI C, and runs in both UNIX and Windows environments (API Documentation). The kind of capability the API provides includes:

- Open up more than one dictionary at a time.
- Parse a sentence with different dictionaries or parsing parameters, and compare the results.
- Limit the time and memory that the parsing process takes.
- Use different "cost functions" for ranking linkages.
- Save linkages from a sentence, and access individual links.

- Extract the domains that links participate in, to perform transformations on a linkage.
- Recover the constituent structure corresponding to a phrase-structure grammar.

3.3.2 Design of a task classifier based on verb's feature vector

The function of the task classifier is to map the user's command to corresponding robot's task category. The task classifier will rely on verbs of a command to carry out classification, some commands without a meaningful verb will be treated as invalid commands and the classifier will ask the user to provide another command. After receiving a command, the classifier will pick out a useful verb and use its feature vector to represent this command; then comparing this vector with the typical task categories' feature vectors and calculating the distance between them a classification will be made according to the available categories. Using this method, a command with a meaningful verb can be successfully classified. Figure 3.1 depicts the schematic diagram of the task classifier.

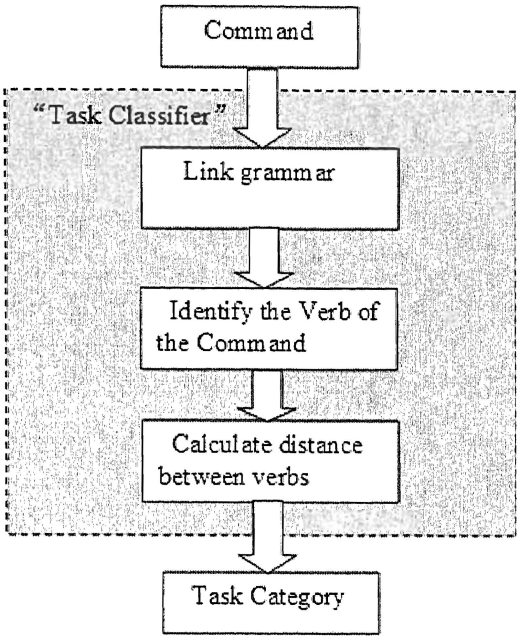


Figure 3.1 Schematic diagram of the task classifier

3.3.3 Implementation of the task classifier

The task classifier is implemented in C++ using Microsoft visual studio 2008. The task classifier employs LGP. The first step of the classifier includes characterising the command, such as syntactical parser, semantic role annotation and most importantly PoS tagging.

After this initial processing, verb, noun, adjective, adverb and other PoS of the commands will be identified. All these words will be stored by the program. The verbs will be used to discover the command's category, whilst the other words will be used to retrieve the hierarchical task tree and find out test actions, which will be discussed in Chapters 4.

The commands usually include more than one verb, for example, the command "could you help me walk upstairs" and command "please help me carry this TV". Figure 3.2(a) and Figure 3.2(b) show the LGP's PoS tagging results of the two commands. The first command has three verbs and the second command has two. The classifier needs to find the useful verb.

(a) (b)

Figure 3.2 The LGP's PoS tagging results

Flow chart of the task classifier is given in Figure 3.3. The first step is initializing all the variables used by task classifier. Then, a user command input is required from the user. NLP is applied after receiving the user's input. Verb's feature vector is then created for the verb contained in the user's input. Based on the obtained feature vector and the keyword in Table 3.1, tasks related to the user's input are identified and classified into one of four typical task categories.

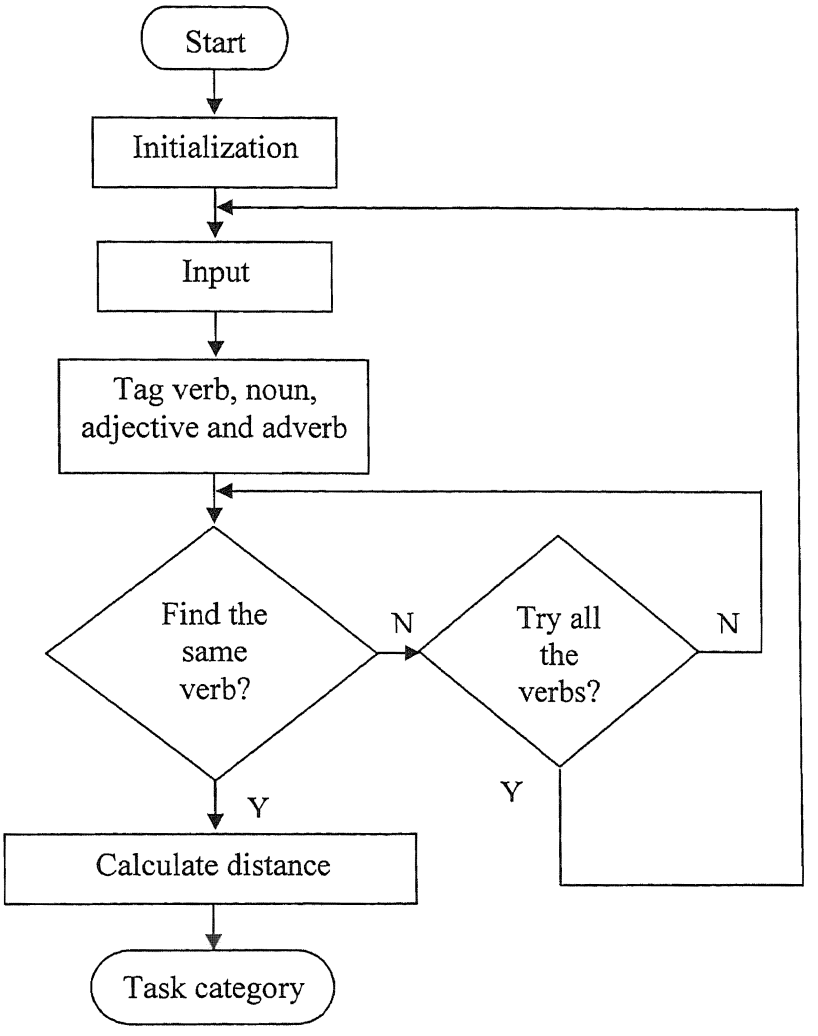


Figure 3.3 Flow chart of task classifier

Figure 3.4 gives an example of the entire work process of the task classifier.

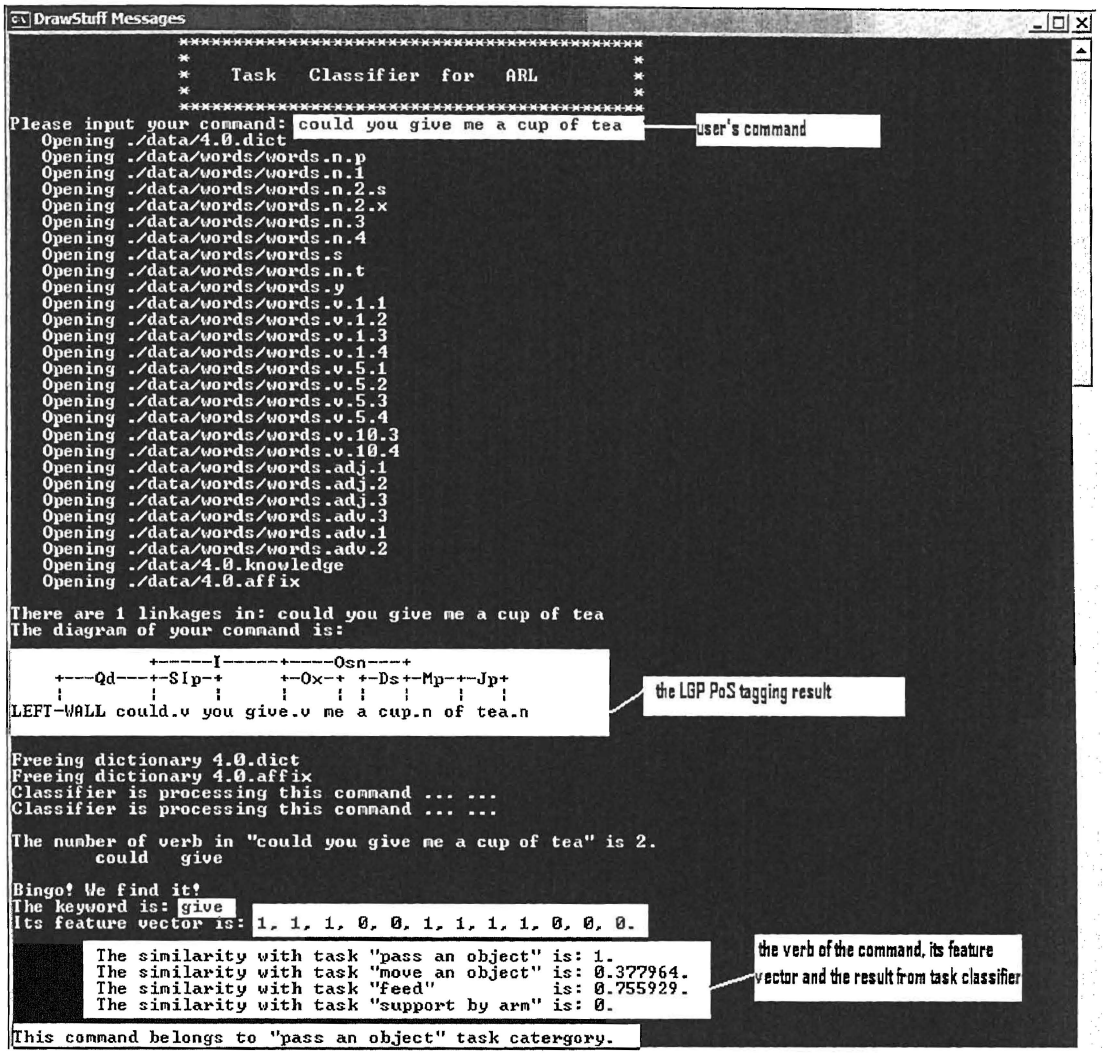


Figure 3.4 Task classifier's work process

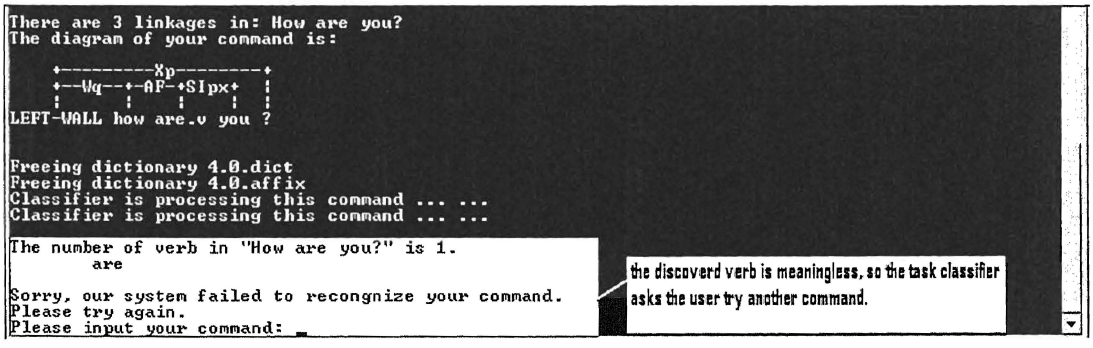


Figure 3.5 Task classifier handles the invalid command

3.3.4 Test results of task classifier

The task classifier is used to process commands corresponding to the following four type tasks:

- ❑ Pass and object
- ❑ Support buy arm
- ❑ Feed the user
- ❑ Help user move an object

As the task classifier relies on verbs of a command to carry out classification, some commands without a meaningful verb will be treated as invalid commands and the classifier will ask the user to provide another command, as shown in figure 3.5.

The commands used for 'pass an object' are:

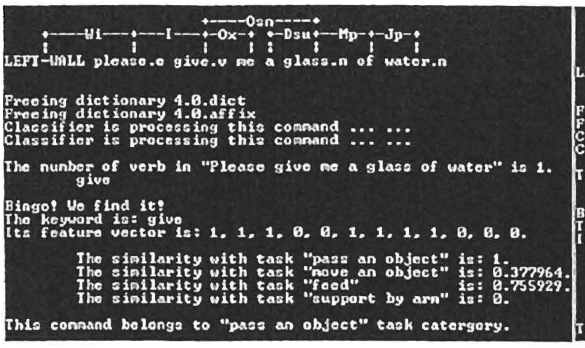
- Please give me a glass of water.
- Can you pass me the newspaper?
- Please bring me a cup of tea.
- Give me an apple.

The task classifier's process results of these commands are shown in Figure 3.6.

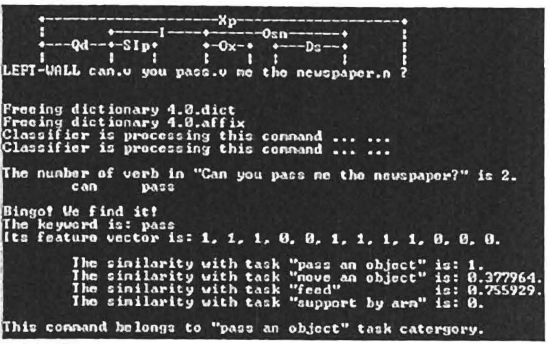
The commands used for 'support by arm' are:

- Could you help me stand up?
- Please help me sit down.
- Help me walk to the upstairs.
- Could you help me walk forward?

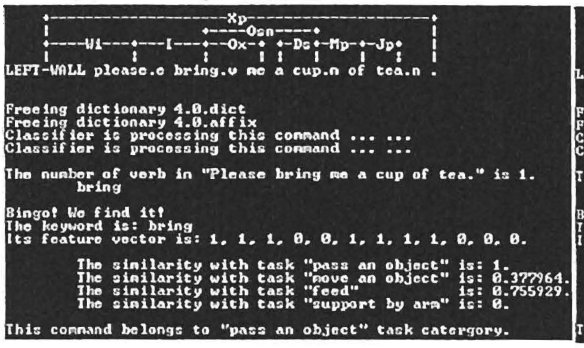
The task classifier's process results of these commands are shown in Figure 3.7.



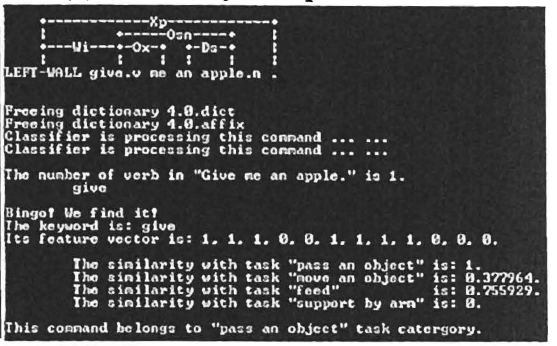
(a) Please give me a glass of water



(b) Can you pass me the

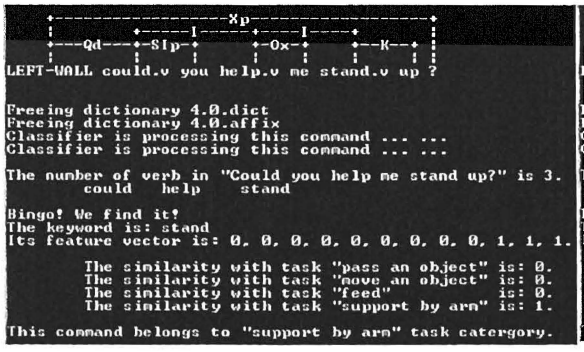


(c) Please bring me a cup of tea.

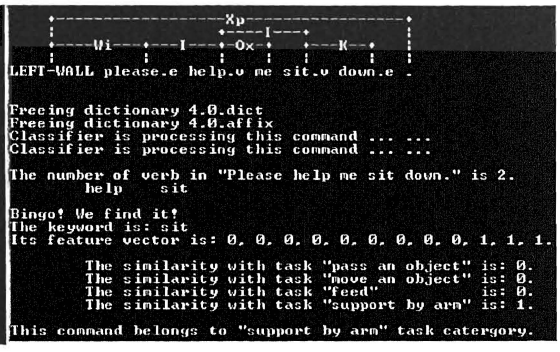


(d) Give me an apple.

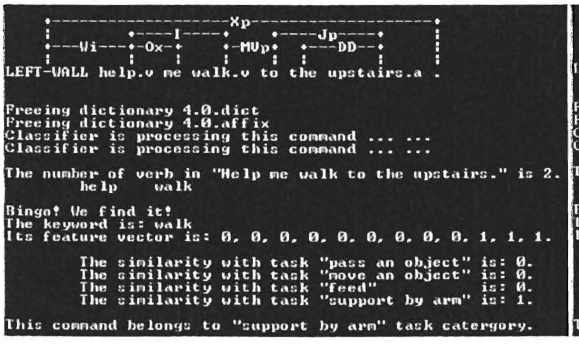
Figure 3.6 Results of commands for 'pass an object'



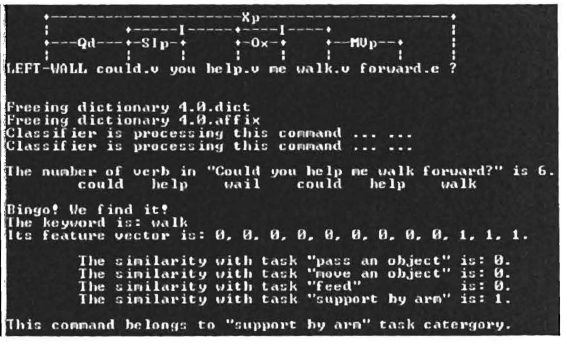
(a)Could you help me stand up?



(b) Please help me sit down.



(c) Help me walk to the



(d)Could you help me walk forward?

Figure 3.7 Results of commands for 'support by arm'

The commands used for 'feed the user' are:

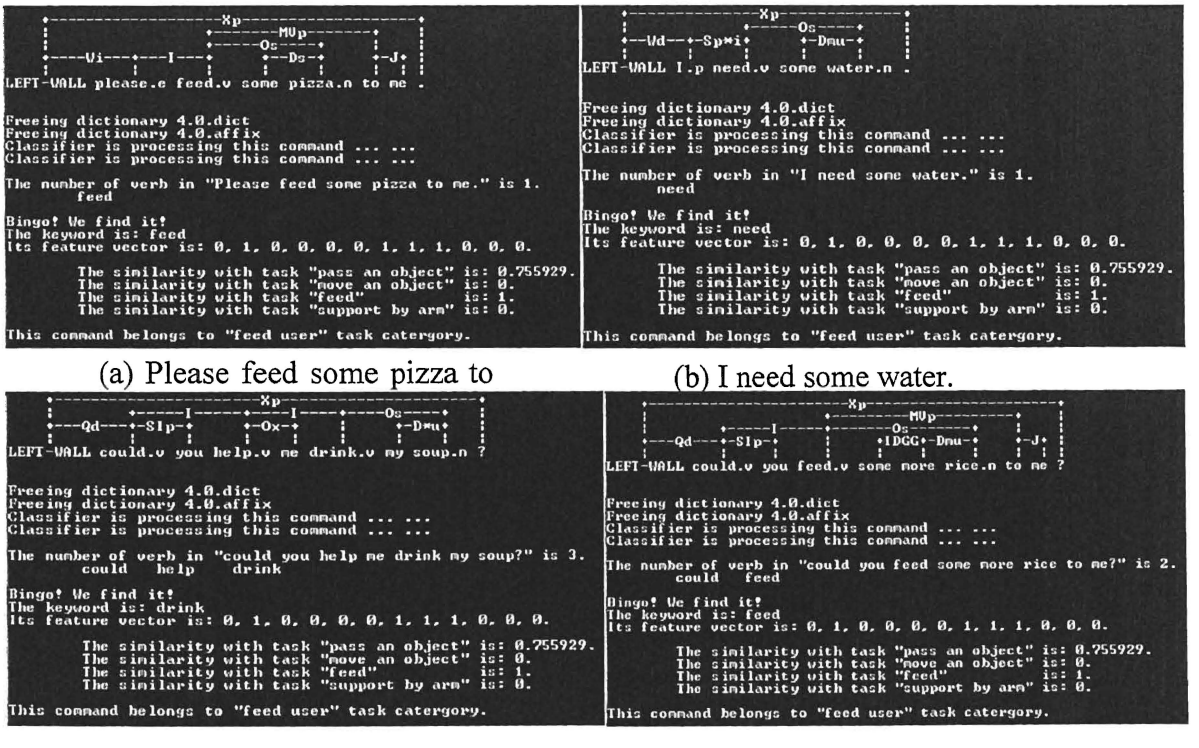
- Please feed some pizza to me.
- I need some water.
- Could you help me drink my soup?
- Could you feed some more rice to me?

The task classifier's process results of these commands are shown in Figure 3.8.

The commands used for 'help user move an object' are:

- Could you help me take this TV?
- Please help me move this bed.
- Help me carry this bucket of water.
- Let us take the fridge together.

The task classifier's process results of these commands are shown in Figure 3.9.



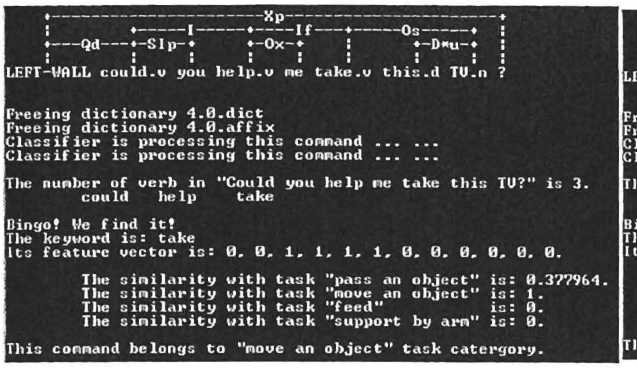
(a) Please feed some pizza to

(b) I need some water.

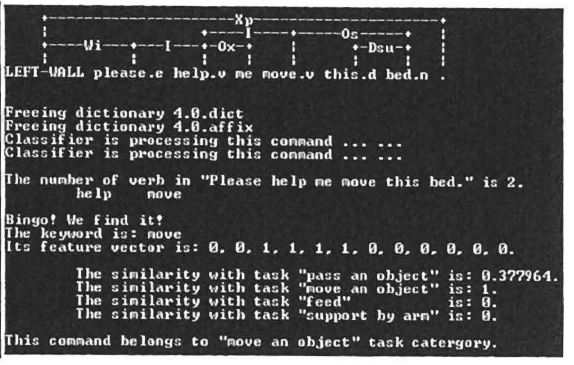
(c) Could you help me drink my

(d) Could you feed some more rice to

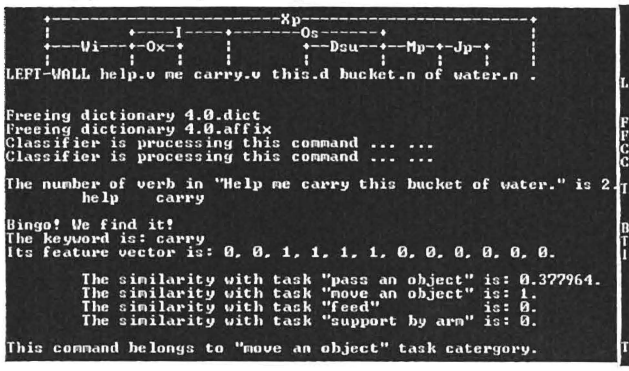
Figure 3.8 Results of commands for 'feed the user'



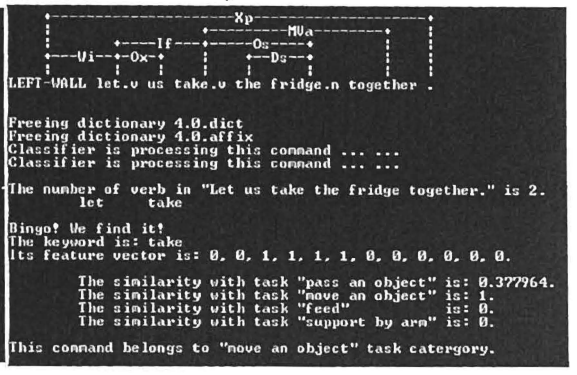
(a) Could you help me take this TV?



(b) Please help me move this bed.



(c) Help me carry this bucket of water.



(d) Let us take the fridge

Figure 3.9 Results of commands for 'help user move an object'

As shown in the above results, using this method, the classifier can eventually pick out a command's useful verb and obtain the verb's feature vector to a carry out task classification. A command with a meaningful verb can be successfully sorted into its corresponding category.

The 12 verbs shown in Table 3.1 cover the most commonly used commands in terms of the four typical tasks. At the current stage of this study, the valid commands refer to commands with the verbs from Table 3.1. The use of these limited verbs to classifier commands, on the other hand, limits the application of this task classifier. This limitation can be resolved by further applying NLP techniques.

CHAPTER 4. TAB STRUCTURE AND DEVELOPMENT

4.1 Representation typical tasks in tree structure

4.1.1 Tree structure

A tree is a way of representing the hierarchical nature of a structure in a graphical form. It is named a 'tree' because the classic representation resembles a tree, even though the chart is generally upside down compared to an actual tree, with the root at the top and the leaves at the bottom. A tree structure contains objects known as NODES and LEAVES. These obey the following rules:

NODES:

- Nodes can exist at any level in the structure.
- A node can have a parent node except if it exists at the top level (level 1), in which case it is known as a ROOT node.
- A tree structure may contain any number of root nodes.
- A parent node must be chosen from those nodes that exist at the level immediately above the current one (i.e.: a node at level 3 can only have a parent from level 2, and so on).
- A node's children can only come from the level immediately below it (i.e.: a node at level 2 can only have children from level 3, and so on).
- A node can have any number of children except those nodes that exist at the bottom level.

LEAVES:

- A leaf can be attached to any node in the structure.
- A leaf represents the end of a branch and can have no children.

A common example of this is the view provided by Windows Explorer which shows the hierarchy of folders and the files contained within those folders (although the structure is displayed from left-to-right rather than from top-to-bottom). In Windows Explorer the nodes are equivalent to folders with the root node representing a driver such as "C:" or "D:", and the leaves are equivalent to files.

Another typical example of a tree structure is a company's organizational hierarchy, as shown in Figure 4.1, where "company", "department" and "section" are the nodes, with "staff" providing the leaves.

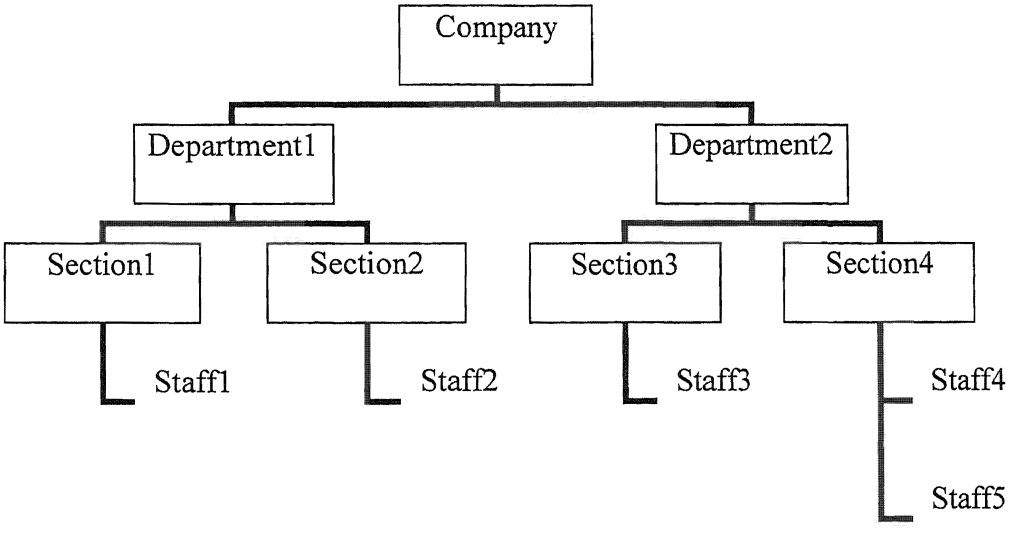


Figure 4.1 A typical tree structure of a company's organizational hierarchy

4.1.2 Tree structure of four typical tasks

In Chapter 3, four typical task categories were proposed as the most useful and worthwhile tasks of domestic service robots. Because a robot can perform numerous similar tasks; the number of test actions for all tasks can be huge while the test actions for the similar tasks can be the same. In this section, every typical task will be expanded into more specific taught tasks, which are represented from the general to the specific.

All the taught tasks will be represented in a hierarchy structure, that is, the top of this hierarchy contains very abstract tasks and the very specific ones are arranged at the bottom. For example, "pass an object" is a highly abstract task, and therefore is arranged on the top. Whilst, "pass a drink", "pass food" and "pass a fruit" are less abstract and because of this they are arranged one level lower than "pass an object". "Pass a cup of tea" and "pass an apple" are more specific so they are arranged at the bottom. The tree structure of "pass an object" is shown in Figure 4.2. The tree structures for "support by arm", "feed the user" and "help user move an object" are shown in Figures 4.3, 4.4 and 4.5, respectively.

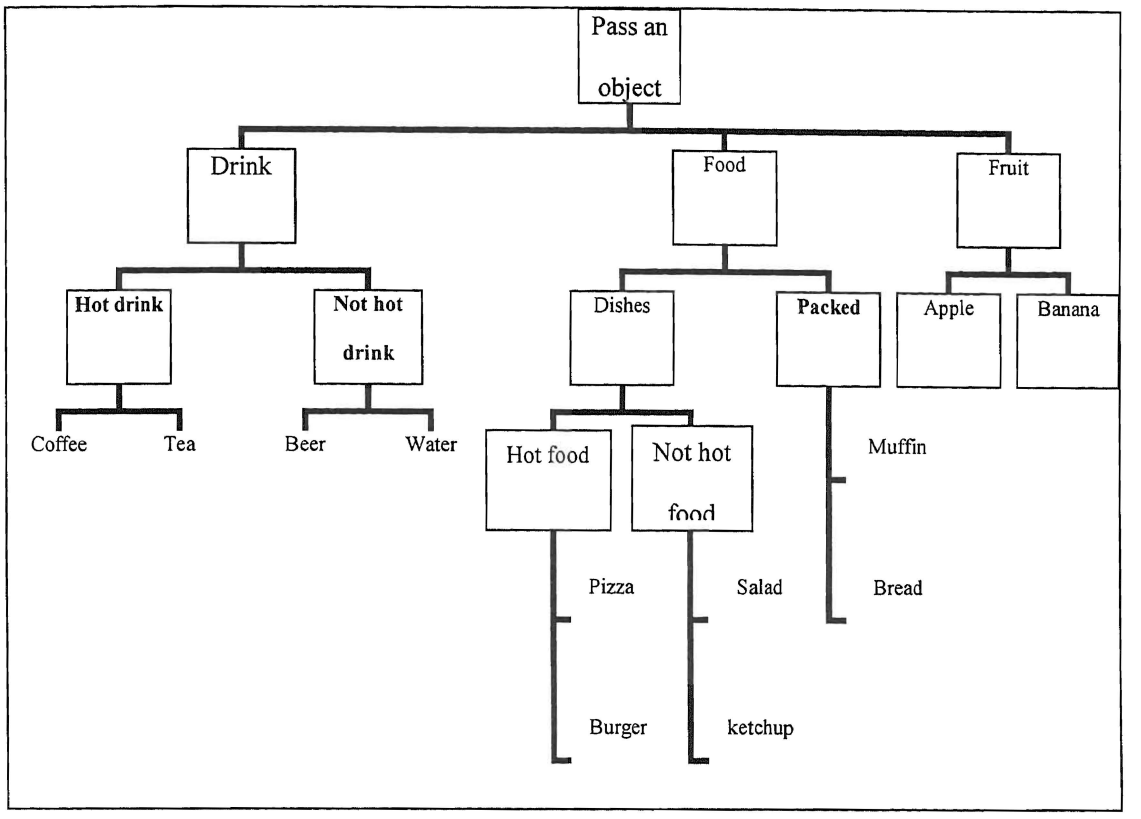


Figure 4.2 Tree structure of 'pass an object' task

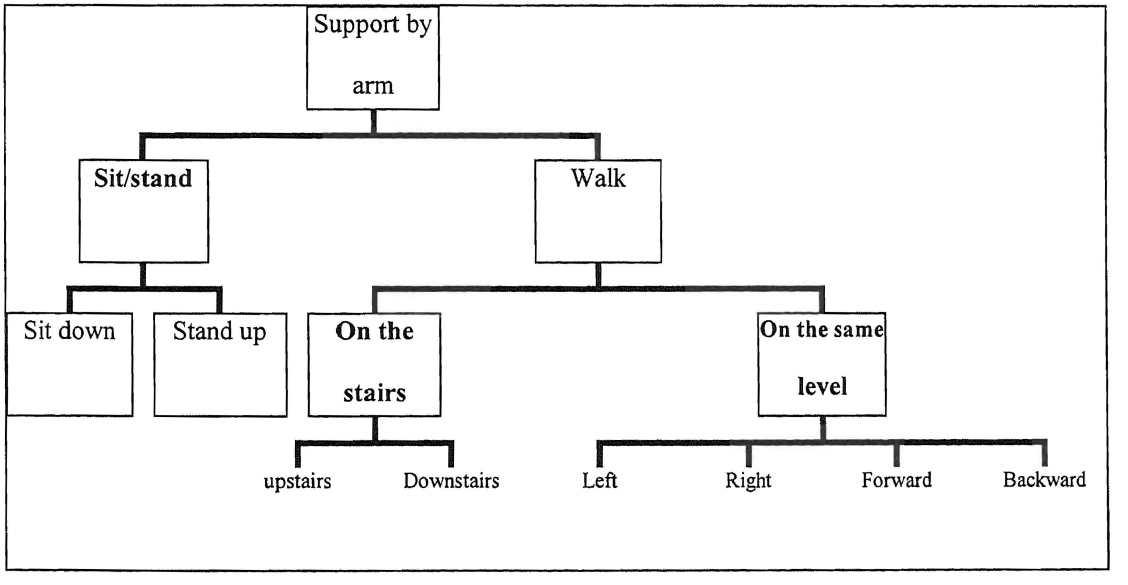


Figure 4.3 Tree structure of 'support by arm' task

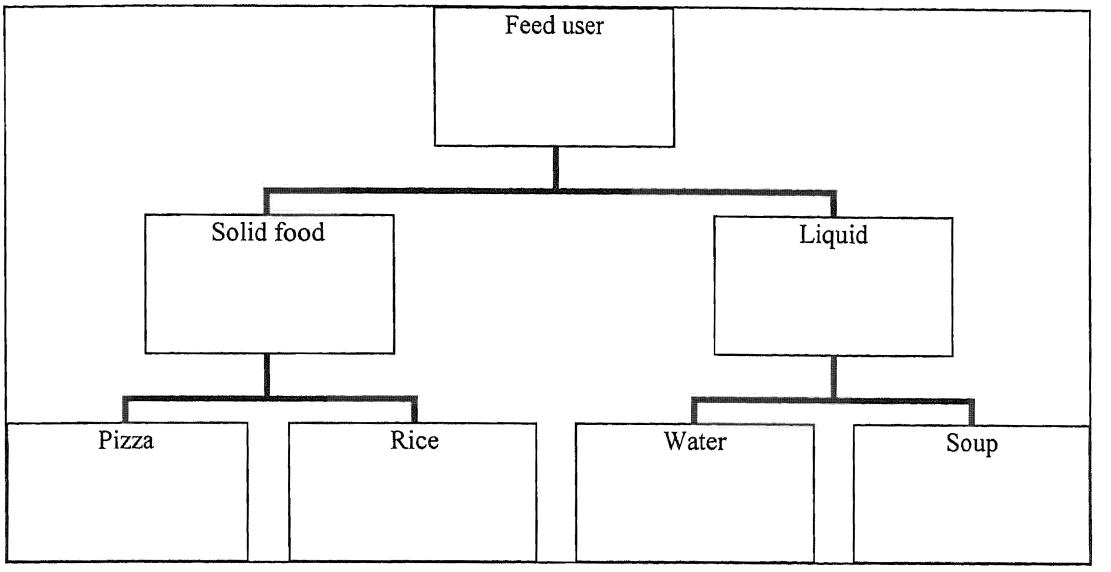


Figure 4.4 Tree structure of 'feed the user' task

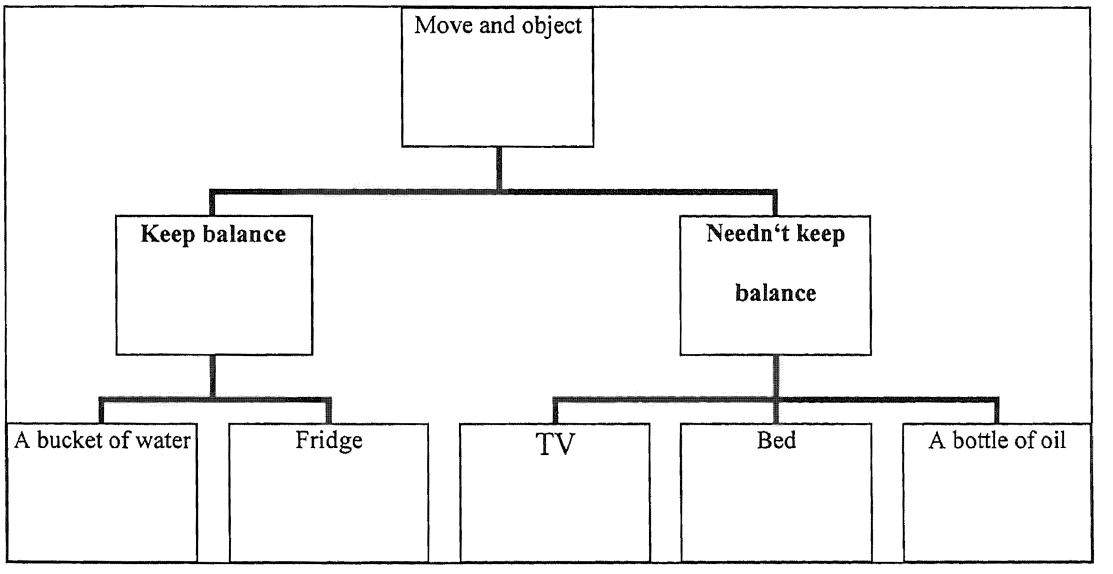


Figure 4.5 Tree structure of 'help user move an object' task

4.2 Implementation of TAB

4.2.1 Relational database

A relational database is a collection of data items organized as a set of formally-described tables from which data can be accessed or reassembled in many different ways without having to reorganize the database tables. The relational database was invented by E. F. Codd at IBM in 1970(E. F. Codd, 1970).

In addition to being relatively easy to create and access, a relational database has the important advantage of being easy to extend. After the original database creation, a new data category can be added without requiring that all existing applications be modified.

A relational database is a set of tables containing data organised into predefined categories. Each table (which is sometimes called a relation) contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns. For example, a typical business order entry database would include a table that described a customer with columns for name, address, phone number, and so forth. Another table would describe an order: product, customer, date, sales price, and so forth. A user of the database could obtain a view of the database that fitted the user's needs. For example, a branch office manager might like a view or report on all customers that had bought products after a certain date. A financial services manager in the same company could, from the same tables, obtain a report on accounts that needed to be paid.

The standard language for a user to access to a relational database is the structured query language (SQL). SQL statements are used both for interactive queries for information from a relational database and for gathering data for reports. The definition of a relational database results in a table of metadata or formal descriptions of the tables, columns, domains, and constraints.

Relational databases have become a predominant choice for the storage of information in new databases used for financial records, manufacturing and logistical information, personnel data and much more. Relational databases have often replaced legacy hierarchical databases and network databases because they are easier to understand and use, even though they are much less efficient. As computer power has increased, the inefficiencies of relational databases, which made them impractical in earlier times, have been outweighed by their ease of use. However, relational databases have been challenged by Object Databases, which were introduced in an attempt to address the object-relational impedance mismatch in relational database, and XML databases.

Relational databases are implemented in relational database management systems. (RDBMS) The three leading commercial relational database vendors are Oracle, Microsoft, and IBM; the three leading open source implementations are MySQL, PostgreSQL, and SQLite.

The RDBMS used to implement TAB is MySQL, which has become the world's most popular open source database because of its consistent fast performance, high reliability and ease of use. MySQL can run on more than 20 platforms including

Linux, Windows, OS/X, HP-UX, AIX and Netware (http://www.mysql.com/why-mysql/).

4.2.2 Store tree structure into relational database

As show in Section 4.1.2, the proposed typical tasks are organized in tree structure to limit the number of test actions and improve the efficiency of ARL. This section explains how to save these tree structures using a relational database such as MySQL. There are two major approaches to storing trees in a relational database: the adjacency list model, and the nested set model (Gijs Van Tulder, 2003).

In the adjacency list model, each item in the table contains a pointer to its parent with the exception of the topmost element, which has a NULL value for its parent. To display or retrieve this tree requires a function, which will start at the root node -- the node with no parent -- and should then deal with all children of that node. For each of these children, the function should retrieve and display all the child nodes of that child. For these children, the function should again handle all children, and so on. This is a regular pattern in the description of this function. We can simply write one function, which retrieves the children of a certain parent node. That function should then start another instance of itself for each of these children, to display all their children. This is the recursive mechanism that gives this model another name recursion method (Joe Celko, 2004).

The adjacency list model has the advantage of being quite simple, easy to use and understand. The downside of this method is that it is slow and inefficient due to the use of recursion.

The second method is commonly referred to as the Nested Set Model (Joe Celko, 2004) in which the hierarchy can be handled as nested containers. Part of Figure 4.3 is pictured this way, as shown in Figure 4.6:

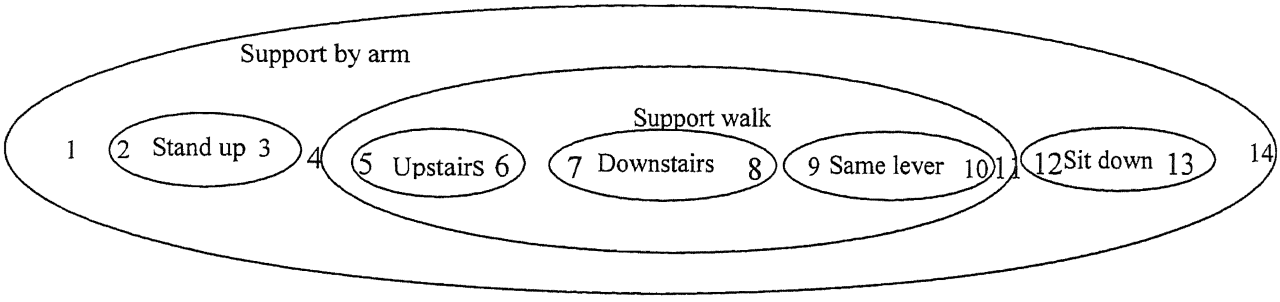


Figure 4.6 Tree structure of 'support by arm' task in the Nested Set Model

Numbering the nodes from the leftmost side and continuing to the rightmost side can determine the left and right values shown in the diagram. These numbers indicate the relationship between each node. Because "upstairs" has the numbers 5 and 6, it is a descendant of the 4-11 "support walk" node. In the same way, we can say that all nodes with left values greater than 1 and right values less than 14, are descendants of 1-14 'support by arm'. The tree structure is now stored in the left and right values. This method of walking around the tree and counting nodes is called the modified preorder tree traversal algorithm (MPTTA) (Gijs Van Tulder, 2003). Using this method, figure 4.2 -4.5 can be represented as figure 4.7-4.10 correspondingly.

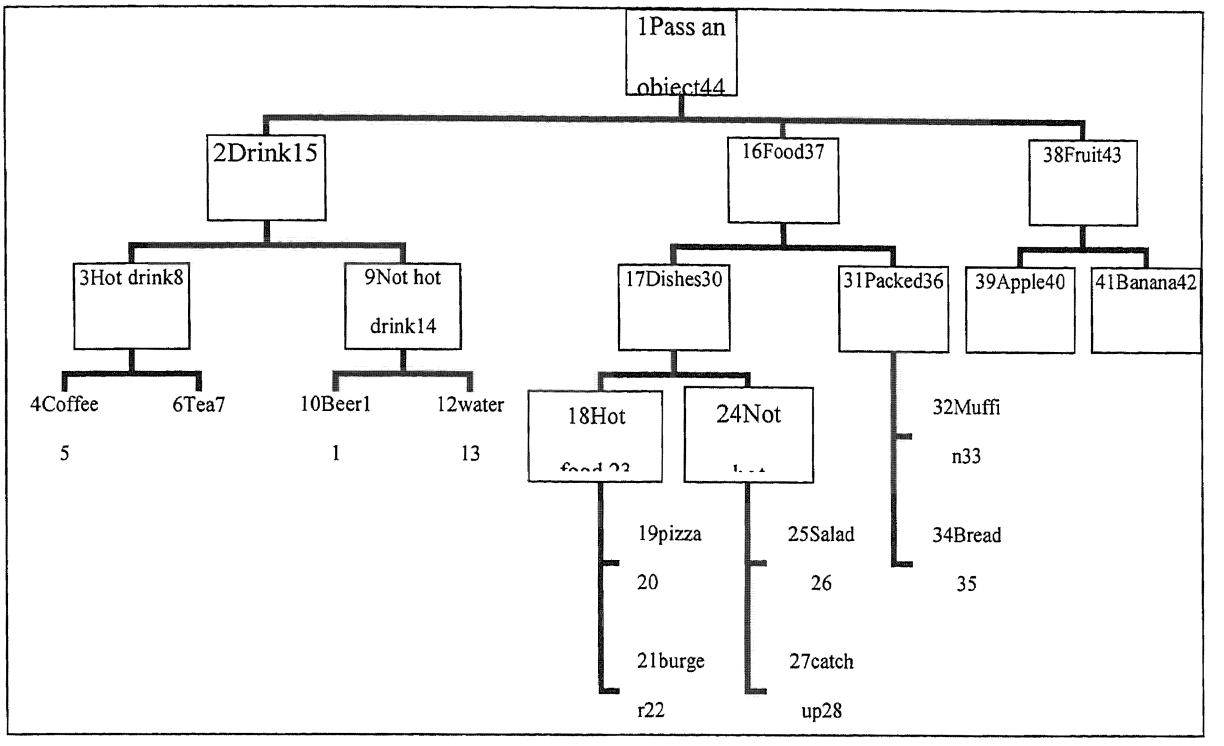


Figure 4.7 Tree structure of 'pass an object' task in MPTTA

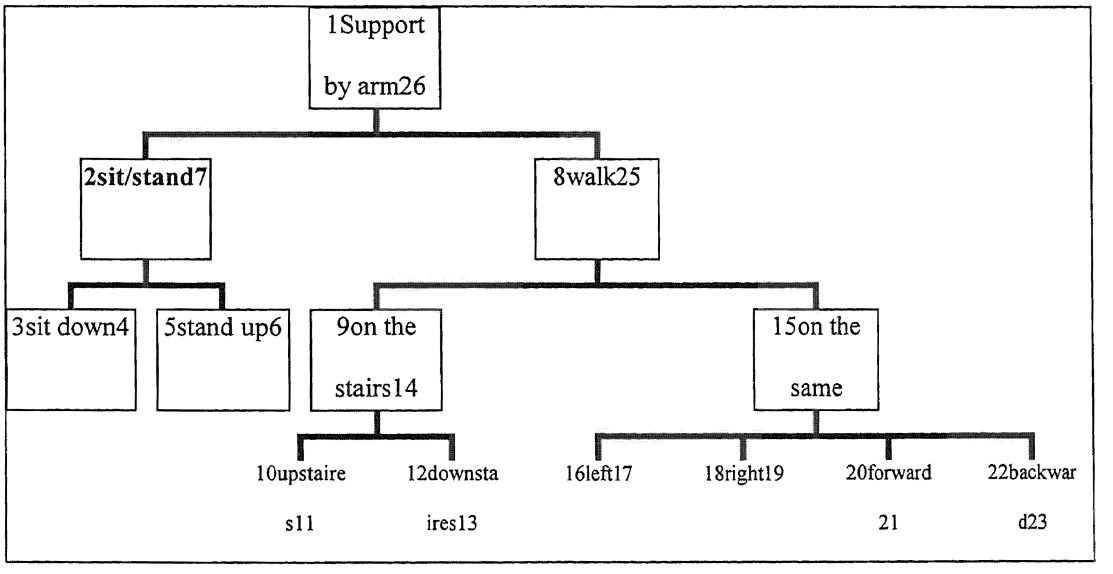


Figure 4.8 Tree structure of 'support by arm' task in MPTTA

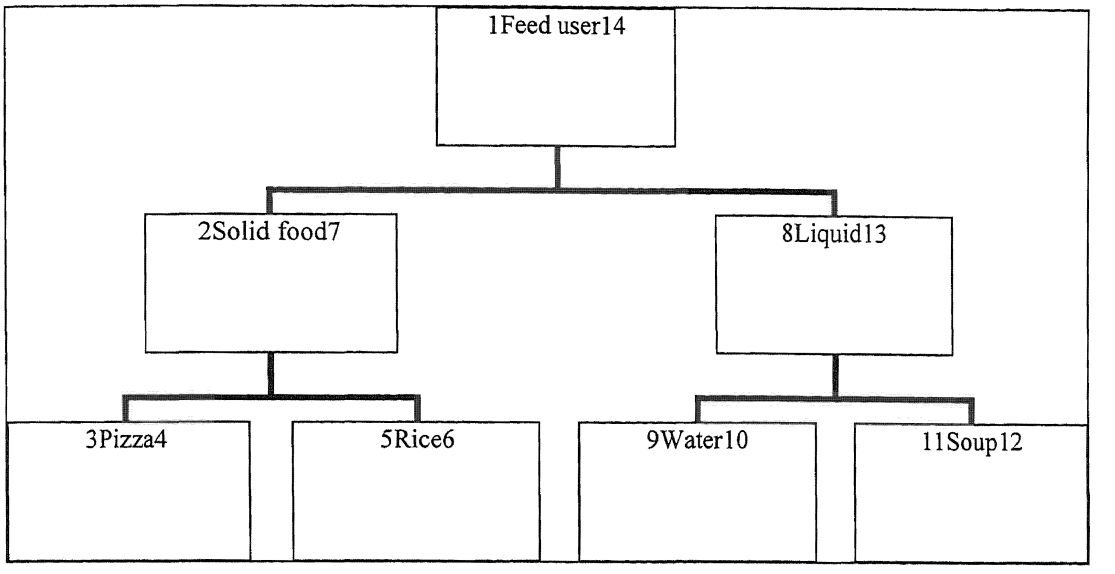


Figure 4.9 Tree structure of 'feed the user' task in MPTTA

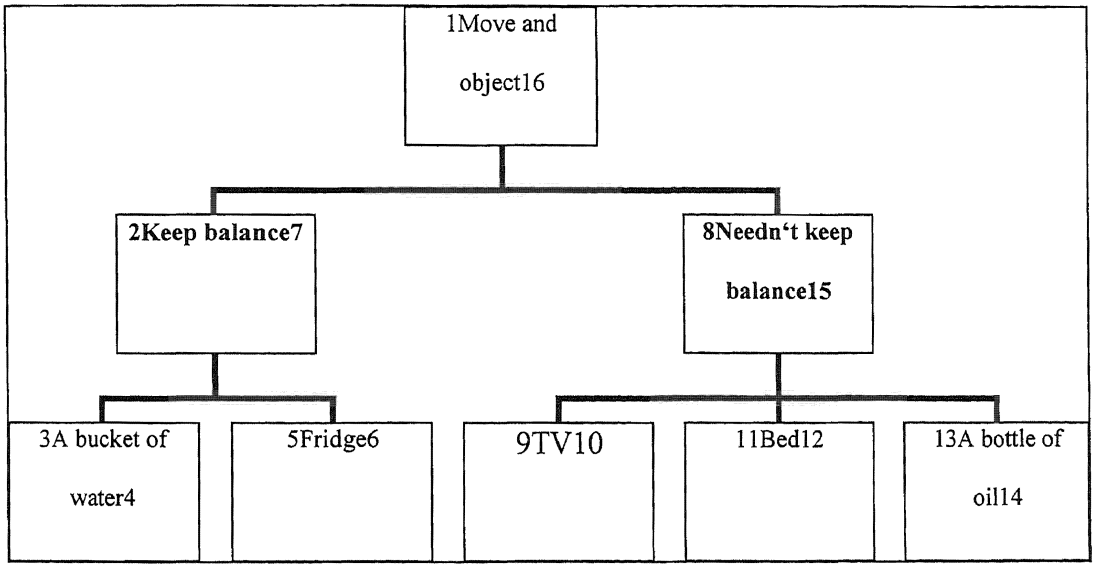


Figure 4.10 Tree structure of 'help user move an object' task in MPTTA

In the following section, the MPTTA algorithm will be used to save the typical tasks into MySQL. These left and right values are useful to show and retrieve task trees: they can be used to display a full tree to show a typical task category from the general to the specific; to retrieve a tree and get the path to a node, this will be used to locate a

particular task in a task tree, which is matched with a command by the task classifier. After finding out the location of a task the next step is obtaining its test actions. The implementation of these manipulations will be discussed in Section 4.3.

4.2.3 TAB's components and implementation

The TAB database is composed of three types of tables, namely "task category", "task tree structure" and "test action". Four typical tasks are stored in the task category table; there are four task tree structure tables to implement the hierarchical structure of four typical tasks and their subtasks correspondingly. All the test actions are stocked in the test action table. Figure 4.11 is an entity-relationship diagram (ERD) to show the relationship between these tables.

An entity-relationship (ER, Peter Pin-shan Chen, 1976) diagram is a specialized graphic that illustrates the interrelationships between entities in a database. ER diagrams often use symbols to represent three different types of information. Boxes are commonly used to represent entities. Diamonds are normally used to represent relationships and ovals are used to represent attributes.

The ERD shown in figure 4.11 contains information of three entities – task category, task tree and test action. There are two relationships between these three entities: "category-tree" and "task-test action". The diagram can distinguish between 1:1 and m:n relationship mappings. The relationship "category-tree" is a 1:1 mapping, that is, one task category may have only one task tree and each task tree will match only one task category. The relationship "task-test action" is an m:n mapping, that is, each task

may have one, two or more test actions and each test action may mach one, two or more tasks.

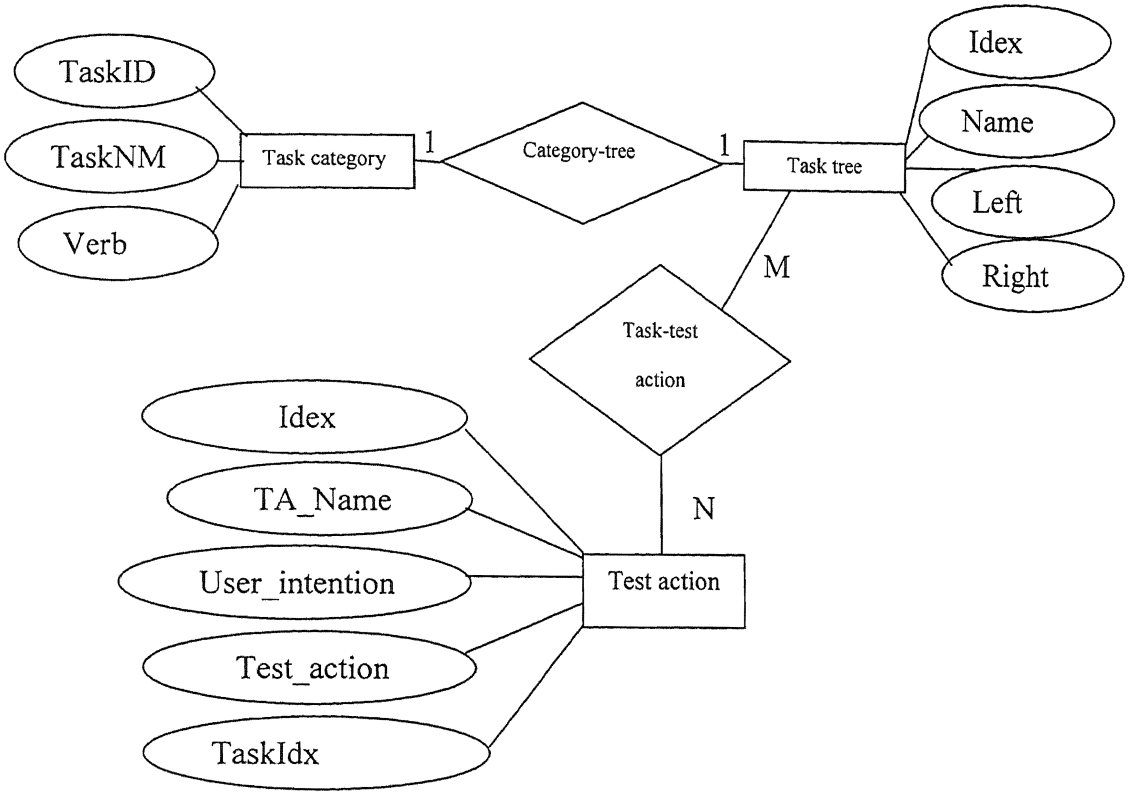


Figure 4.11 The entity relationship diagram (RED) of TAB

Using the following SQL language to establish task category table:

```
DROP TABLE IF EXISTS task_category;
CREATE TABLE task_category
(
        TaskID    int(10) not null,
        TaskNm    VARCHAR(30) not null,
        verb VARCHAR(30)
 );
```

Dumping data for table "task_category" using the following commands:

INSERT INTO `task_category` VALUES (1,'move object' ,'move');

INSERT INTO `task_category` VALUES (2,'pass object','pass');

INSERT INTO `task_category` VALUES (3,'feed food','feed');

INSERT INTO `task_category` VALUES (4,'support by arm','support');

Figure 4.12 provides information about the columns in "task_category" table and Figure 4.13 shows the entire contents of this table.

```
mysql> describe task_category;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| TaskID | int(10)     | NO   |     | NULL    |       |
| TaskNm | varchar(30) | NO   |     | NULL    |       |
| verb   | varchar(30) | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+
```

Figure 4.12 Columns of the tasks' category table

TaskID	TaskNm	verb
1	move object	move
2	pass object	pass
3	feed food	feed
4	support by arm	support

Figure 4.13 Contents of the tasks' category table

All the four typical taught tasks are stored in the task_category table, with three attributes: tasks' identifier (TaskID), tasks' name (TaskNm) and tasks' keyword (verb). This table is used for task classification, that is, the task classifier introduced in Chapter 3 will use information in this table to process use's commands and determine their corresponding task categories. This task classification process will follow the workflow shown in Figure 3.3. After the task classifier identifies the verb

of a command, the verb that is the keyword of the command is used to find out to which task category the command belongs to using the task category table. As the categories in the task category table are too abstract to be matched with key test actions. Therefore, more specific tasks will be determined from one of these four task tree structure tables.

Figures 4.14 to 4.17 shows the contents of four tables, these tables are used to store four typical tasks and their more specific subtasks in a hierarchical structure. The "Idex" columns are for tasqks' identifiers; the "name" columns are brief descriptions of tasks; the "lft" and "rgt" columns are the values for representing the tree structure, see preceding Section 4.4.2 for the meanings and details about these values.

Idex	name	lft	rgt
2.2.1	Pass an object	1	44
2.2.2	drink	2	15
2.2.3	food	16	37
2.2.4	fruit	38	43
2.2.5	Hot drink	3	8
2.2.6	Not hot drink	9	14
2.2.7	dishes	17	30
2.2.8	packed	31	36
2.2.9	apple	39	40
2.2.10	banana	41	42
2.2.11	coffee	4	5
2.2.12	tea	6	7
2.2.13	beer	10	11
2.2.14	water	12	13
2.2.15	Hot food	18	23
2.2.16	Not hot food	24	29
2.2.17	Muffin	32	33
2.2.18	bread	34	35
2.2.19	pizza	19	20
2.2.20	burger	21	22
2.2.21	salad	25	26
2.2.22	ketchup	27	28

Figure 4.14 Tree structure of "pass an object" task

Idex	name	lft	rgt
2.1.1	Support by arm	1	26
2.1.2	Stand/sit	2	7
2.1.3	walk	8	25
2.1.4	sit down	3	4
2.1.5	Stand up	5	6
2.1.6	On the stairs	9	14
2.1.7	On the same level	15	24
2.1.8	upstairs	10	11
2.1.9	downstairs	12	13
2.1.10	Left	16	17
2.1.11	Right	18	19
2.1.12	Forward	20	21
2.1.13	Backward	22	23

Figure 4.15 Tree structure of "support by arm" task

Idex	name	lft	rgt
2.4.1	Feed user	1	14
2.4.2	Solid food	2	7
2.4.3	Liquid	8	13
2.4.4	Pizza	3	4
2.4.5	Rice	5	6
2.4.6	Water	9	10
2.4.7	soup	11	12

Figure 4.16 Tree structure of "feed the user" task

Idex	name	lft	rgt
2.3.1	Move an object	1	16
2.3.2	Keep balance	2	7
2.3.3	Needn't keep balance	8	15
2.3.4	A bucket of water	3	4
2.3.5	fridge	5	6
2.3.6	TV	9	10
2.3.7	Bed	11	12
2.3.8	table	13	14

Figure 4.17 Tree structure of "help user move an object" task

As the task tree is from the general to the specific, this special structure is very useful for finding a task's corresponding test actions. All test actions are stored in the test action table, as shown in Figure 4.18.

idex	TA_name	user_intention	Test_actions	TaskIdx
1	Walk forward	Walk forward	Forward a little	2.1.1, 2.1.3, 2.1.7¶
2	Walk backward	Walk backward	Back a bit	2.1.1, 2.1.3, 2.1.7¶
3	Turn right	Turn right	Turn right a little	2.1.1, 2.1.3, 2.1.7¶
4	Turn left	Turn left	Turn left a little	2.1.1, 2.1.3, 2.1.7¶
5	Walk upstairs	Walk upstairs	Support the user walk up, then walk up	2.1.1, 2.1.3, 2.1.6¶
6	Walk downstairs	Walk downstairs	Walk down first, then support the user walk down	2.1.1, 2.1.3, 2.1.6¶
7	Support stand up	Stand up	Hold the user's arm and help him stand up	2.1.1, 2.1.2¶
8	Support sit down	Sit down	Hold the user's arm and help him sit down	2.1.1, 2.1.2¶
9	Take up	Hold firmly	Take up	2.2.1, 2.2.5, 2.2.15¶
10	Take back	Hold firmly	Tack back a bit	2.2.1, 2.2.5, 2.2.6, 2.2.15, 2.2.16¶
11	Put down	Ready to hold	Put down bit by bit	2.2.1, 2.2.4,2.2.5, 2.2.6, 2.2.8, 2.2.15, 2.2.16¶
12	Release	Ready to hold	Release a bit	2.2.1,2.2.4,2.2.5, 2.2.6, 2.2.8,2.2.15, 2.2.16¶
13	Move up	Move up	Move up	2.3.3¶
14	Move down	move down	Move down a bit	2.3.3¶
15	Move left	left	Move left a bit	2.3.1, 2.3.2, 2.3.3¶
16	Move right	right	Move right a bit	2.3.1, 2.3.2, 2.3.3¶
17	Move forward	forward	Move forward a bit	2.3.1, 2.3.2, 2.3.3¶
18	Move backward	backward	Move backward a bit	2.3.1, 2.3.2, 2.3.3¶
19	Stay still	Stay still	Stay still a bit	2.3.1, 2.3.2, 2.3.3¶
20	Move near user's mouth	Need more	Move near user's mouth	2.4.1¶

Figure 4.18 Test action table

The identifiers of tasks using a specific test action will be stored into the "TaskIdx" column; this column can be used to find out a task's corresponding test actions, querying this column can retrieve corresponding test actions for a particular task; section 4.3.2 will give more details about retrieving test actions.

4.3 Retrieval of test actions

The aim of arranging tasks into a tree structure is to allow the same test action can be used in similar tasks. As shown in Figure 4.2, there are totally 12 specific tasks in the "pass an object" task category, and only four test actions for them in the test action

table. Because all these tasks belong to the same category, some of them can share the same test actions.

4.3.1 Locating a task in task tree

When devising the architecture of task trees for these typical tasks, the aim is to place similar tasks in the same tree, with the most general task at the top and the most specific tasks at the bottom as leaf nodes. Take figure 4.2 for example, the tasks requiring the same test actions are at the same level on the task tree, such as 'hamburger' and 'pizza', they represent task "give me a hamburger ", "give me some pizza" correspondingly, and are at the same level; but not all the tasks on the same level use the same test actions, both the 'salad' and 'ketchup' are on the same level as "pizza", but they require different test actions, because the "salad" and "ketchup" belong to the "not hot dishes" task category while "pizza" belongs to the "hot dishes" category.

Therefore, before identifying the test actions for a task, we should know the task's location in a task tree structure. As all the task trees are stored in database using MPTTA, after the task is located, its left and right values can be used to get its ancestor, as these values are between its ancestors' corresponding values; both the task and its ancestors are nodes of a task tree, these nodes can be used to established a path to this task. To locate a task is to find out a list of all ancestors of that task.

An example of locating task in a task tree is task "give me some pizza", as shown in the task tree structure in figure 4.7, the clue of this task is "pizza", its left and right

value is "19" and "20"; this task belongs to the "pass an object" task tree, the table to store this task tree is called as "pass_tree_structure", as shown in figure 4.14; using the following SQL commands we can obtain the location and get the path to this task:

Command1:

```
SELECT name FROM pass_tree_structure WHERE lft<=19 and rgt>=20
```

Command2:

```
SELECT parent.name
FROM pass_tree_structure as parent, pass_tree_structure as node
WHERE node.lft BETWEEN parent.lft AND parent.rgt
AND node.name='pizza'
order by parent.lft
```

As an option, we use command2; the advantage of the second command is that it does not use left and right value explicitly. Figure 4.19 shows the path to a specific task, with the general task on the above and the specific task below.

```
mysql> SELECT name FROM pass_tree_structure WHERE lft<=19 and rgt>=20;
+----------------+
| name           |
+----------------+
| Pass an object |
| food           |
| dishes         |
| Hot food       |
| pizza          |
+----------------+
5 rows in set (0.00 sec)
```

(a) The result of command1

```
mysql> SELECT parent.name
    -> FROM  pass_tree_structure as parent,pass_tree_structure as node
    -> WHERE node.lft BETWEEN parent.lft AND parent.rgt
    -> AND node.name='pizza'
    -> order by parent.lft
    -> ;
+----------------+
| name           |
+----------------+
| Pass an object |
| food           |
| dishes         |
| Hot food       |
| pizza          |
+----------------+
5 rows in set (0.05 sec)
```

(b) The result of command2

Figure 4.19 The path to a specific task

During this task locating process, "pizza" is chosen to represent the task "give me some pizza", using this idea, we find a solution to locate the task in a task tree. That is, find a word as a task's clue, and store this clue instead of the task in database to represent the corresponding task. Actually, these task trees shown in figure 4.14-4.17 are described using this method. In the current stage of this study, the chosen task's clue is noun from a command.

As another example to show how to locate a task, figure 4.20 shows the path to task "bring me some salad".

```
mysql> SELECT parent.name
    -> FROM pass_tree_structure as parent, pass_tree_structure as node
    -> WHERE node.lft BETWEEN parent.lft AND parent.rgt
    -> AND node.name='salad'
    -> order by parent.lft;
+----------------+
| name           |
+----------------+
| Pass an object |
| food           |
| dishes         |
| Not hot food   |
| salad          |
+----------------+
5 rows in set (0.00 sec)
```

Figure 4.20 The path to a task "bring me some salad"

Comparing with figure 4.19, these two tasks "give me some pizza" and "bring me some salad" share the same task path, composed of three tasks from very general task to specific one: "pass an object", "pass food" (its clue is "food") and "pass dishes" (its clue is "dishes"). However, the fourth component of the path of task "give me some pizza" is "hot food", while the fourth component of "give me some salad" is "not hot food". This difference will determinate that two tasks require different test actions, see figures 4.21 and 4.22.

Getting the location and finding the path of a task is crucial to the retrieval of test actions, which will be depict in the next section.

4.3.2 Backward retrieval of test actions

Generally, different tasks determinate and require different test actions. The task classifier firstly finds a command's category and this command's clue, which will be used to locate a task and its path from the corresponding task tree. All the tasks are represented from very general one to very specific one in a task tree, so is the path to a task, it also has the general on top and the specific at the bottom. The retrieval of a task's test actions should from the specific task to the general task, which is from the bottom to the top of a task tree, and this is why we call this method: backward retrieval.

Using the task's location and path information, the test action retrieval process will begin from the last node (the leaf and bottom task) of this path, and will continue

from bottom specific task to top general task until a task is particular and precise enough and its test actions can be gotten from test actions table.

Figure 4.21 shows the task tree of typical task "pass an object", this task tree is described in figure 4.14; the difference is figure 4.21 is from the very abstract task to the very specific task.

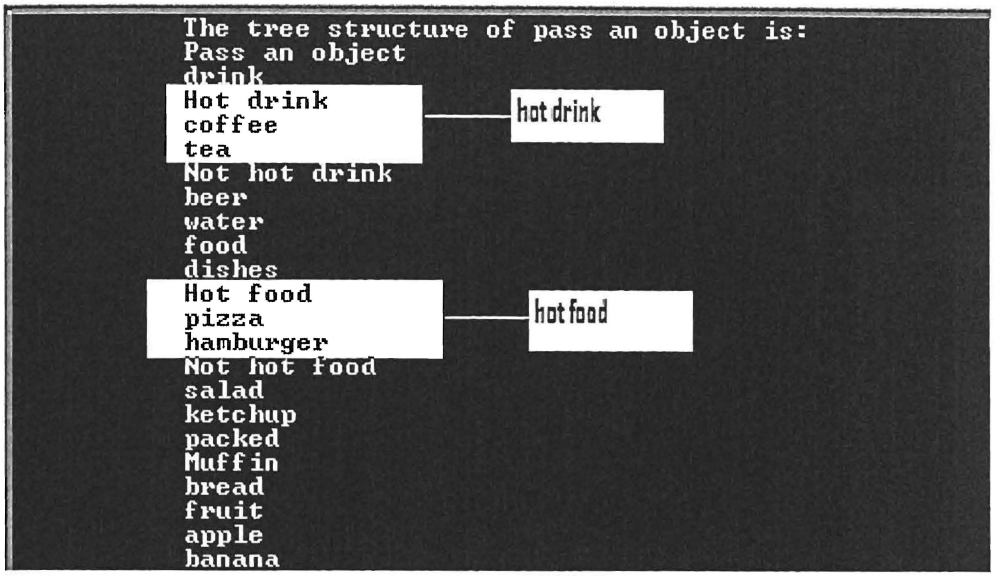


Figure 4.21 The task tree of "pass an object"

As "give me some pizza" and "pass me some salad" blond to "pass an object", therefore, the two task's path, shown in figure 4.19 and figure 4.20, can be derived from figure 4.21. The same as task "please give me a cup of tea" and task "please pass me a Hamburger", as tea belongs to "hot drink" and hamburger belongs to "hot food".

Take task "please give me a hamburger" for example, the test action retrieval result for this task is shown in figure 4.22.

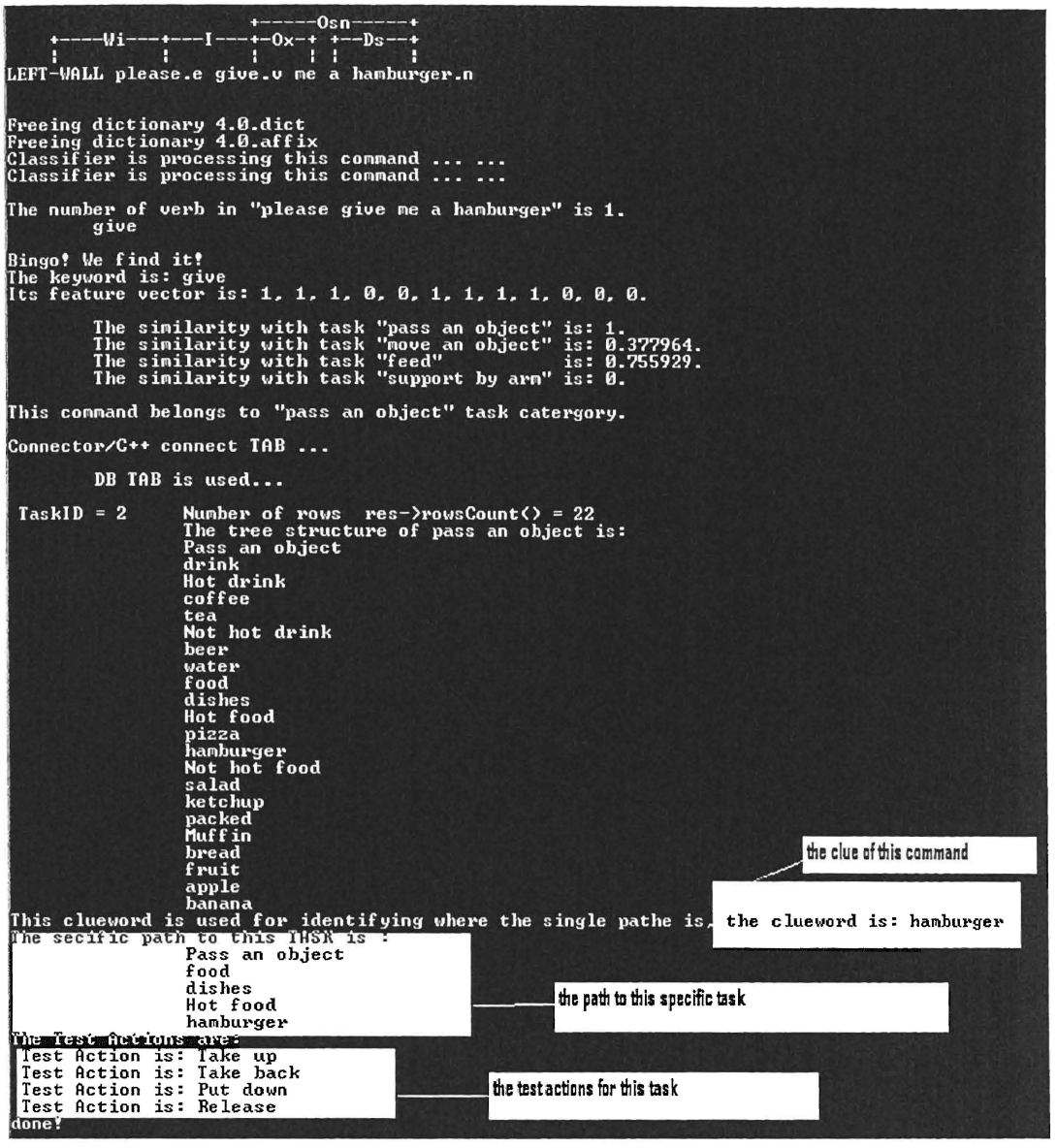


Figure 4.22 The retrieval of test actions

The noun "hamburger" is used as a clue for this task, as in the result, this clue can successfully locating its corresponding task and finding the path to this task; at last, the four test actions used for this task are retrieved from TAB, they are: task up, task back, put down and release.

Figure 4.23 shows the test actions for "give me some pizza" and "bring me some salad", as mentioned above; these two tasks have different test actions. However, "give me some pizza" and "give me a hamburger" share the same test actions, as both of them belong to the "pass hot foot" task category.

```
This clueword is used for identifying where the single pathe is, the clueword is: pizza
The secific path to this TASK is :
                Pass an object
                food
                dishes
                Hot food
                pizza
The Test Actions are:
 Test Action is: Take up
 Test Action is: Take back
 Test Action is: Put down
 Test Action is: Release
done!
```

(a) Test actions for "give me some pizza"

```
This clueword is used for identifying where the single pathe is, the clueword is: salad
The secific path to this TASK is :
                Pass an object
                food
                dishes
                Not hot food
                salad
The Test Actions are:
 Test Action is: Take back
 Test Action is: Put down
 Test Action is: Release
done!
```

(b) Test actions for "bring me some salad"

Figure 4.23 Other results for the retrieval of test actions

This test action retrieval method will be used in all four typical tasks to find corresponding test actions, the rest results will be discussed in chapter 6.

CHAPTER 5. INTEGRATION AND SIMULATUON

5.1 Tool kit used for simulation

The tool kit used for simulation for this study is Open Dynamics Engine (ODE), which is a free, industrial quality library for simulating articulated rigid body dynamics. ODE is good for simulating ground vehicles, legged creatures, and moving objects in virtual reality environments. It is fast, flexible and robust, and it has built-in collision detection.

5.1.1 Introduction

ODE is being developed by Russell Smith with help from several contributors (User guider, V0.5). This library is free software can be redistributed and/or modified under the terms of either the GNU Lesser General Public License or the BSD-style license.

Some of ODE's features are:

- Rigid bodies with arbitrary mass distribution.
- Joint types: ball-and-socket, hinge, slider (prismatic), hinge-2, fixed, angular motor, universal.
- Collision primitives: sphere, box, capped cylinder, plane, ray, and triangular mesh.
- Collision spaces: Quad tree, hash space, and simple.

- Simulation method: The equations of motion are derived from a Lagrange multiplier velocity based model due to Trinkle/Stewart and Anitescu/Potra.
- A first order integrator is being used. It's fast, but not accurate enough for quantitative engineering yet. Higher order integrators will come later.
- Choice of time stepping methods: either the standard ``big matrix" method or the newer iterative Quickstep method can be used.
- Contact and friction model: This is based on the Dantzig LCP solver described by Baraff, although ODE implements a faster approximation to the Coloumb friction model.
- Has a native C interface (even though ODE is mostly written in C++).
- Has a C++ interface built on top of the C.
- Many unit tests and more being written all the time.
- Platform specific optimizations.

5.1.2 Installing and using ODE

ODE is Open source software, after downloading the source code from ODE's official website: http://www.ode.org/download.html, using the automatic project generation tool Premake build system to generate project files. Premake is available from http://premake.sourceforge.net.

To create a set of custom project files, first run Pemake on the command prompt, then type "premake --help" to see the available options. To generate the new project files using the command as:

premake [options] --target [toolset]

To build the demo applications with drawstuff library for MS visual studio 2008, using the following commands:

premake --with-demos --target vs2008

The directory "ode/build" contains appropriate solution for the Visual Studio 2008 version project files. Open and build the solution, the Static Library (Lib) and Shared Library (DLL) can be gotten; the demos and the programs at the "test/example" are the best way to understand how to use ODE.

Source files that use ODE only need to include a single header file "ode/ode.h"; to use the drawstuff library an additional head file "drawstuff/drawstuff.h" is needed.

5.1.3 Concepts and data types

ODE simulation mechanism has two main components: a dynamics simulation engine and a collision detection engine. In ODE two fundamental concepts are used: rigid body and geometry object (or "geom" for short). Rigid body applies to dynamics and has various dynamical properties, such as mass and velocity; geometry object applies to collision and represents a rigid shape, like box and cylinder. To use the collision engine in a rigid body simulation, geoms are associated with rigid body objects. This allows the collision engine to get the position and orientation of the geoms from the bodies. A body and a geom together represent all the properties of the simulated

object. During ODE's simulation process, the collision engine is given information about the shape of each body; this shape is represented by a geometry class, such as box, sphere, cylinder, etc. The shape of a rigid body is not a dynamical property, only collision detection that cares about the detailed shape of the body. At each simulation time step the collision engine figures out which bodies touch each other and passes the resulting contact point information to the user. The user in turn creates contact joints between bodies. Section 5.2 will give more details about collision and contact during the simulation process.

To simulate a reality world, a lot a concepts and data types are needed to represent objects, here only introducing some basic ones to give a general description of ODE.

There are some basic kinds of object that can be created by ODE:

- dBody - a rigid body.
- dWorld - a dynamics world.
- dGeom - geometry (for collision).
- dSpace - a collision space.
- dJoint - a joint
- dJointGroup - a group of joints.

A rigid body has various properties from the point of view of the simulation. Some properties change over time:

- Position vector (x,y,z) of the body's point of reference, the point of reference must correspond to the body's center of mass.

- Linear velocity of the point of reference, a vector (vx,vy,vz).
- Orientation of a body, represented by a quaternion (qs,qx,qy,qz) or a 3x3 rotation matrix.
- Angular velocity vector (wx,wy,wz) which describes how the orientation changes over time.

Other body properties are usually constant over time:

- Mass of the body.
- Position of the center of mass
- Inertia matrix. This is a 3x3 matrix that describes how the body's mass is distributed around the center of mass.

The world object is a container for rigid bodies and joints. Objects in different worlds can not interact, for example rigid bodies from two different worlds can not collide. All the objects in a world exist at the same point in time, thus one reason to use separate worlds is to simulate systems at different rates. Most applications will only need one world.

Geometry objects are the fundamental objects in the ODE collision system. A geom can represents a single rigid shape (such as a sphere or box), or it can represents a group of other geoms --a special kind of geom called a "space". Every geom is an instance of a class, such as sphere, plane, or box. There are a number of built-in classes in ODE.

Geoms can be placeable or non-placeable. A placeable geom has a position vector and a 3*3 rotation matrix, just like a rigid body, that can be changed during the simulation. A non-placeable geom does not have this capability - for example, it may represent some static feature of the environment that can not be moved. A space is a non-placeable geom that can contain other geoms. It is similar to the rigid body concept of the "world", except that it applies to collision instead of dynamics. Space objects exist to make collision detection go faster.

In real life a joint is something like a hinge, which is used to connect two objects. In ODE a joint is very similar: It is a relationship that is enforced between two bodies so that they can only have certain positions and orientations relative to each other. This relationship is called a constraint -- the words joint and constraint are often used interchangeably. Figure 5.1 shows three different constraint types in ODE.

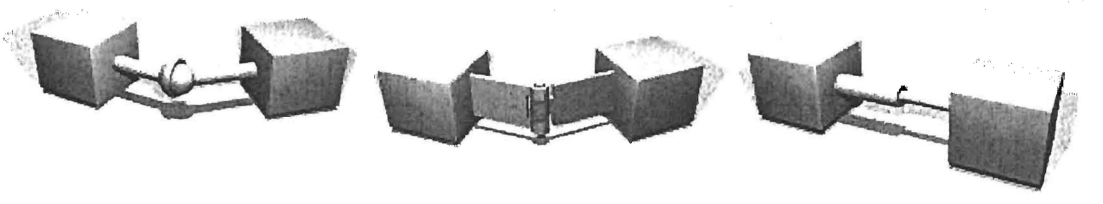


Figure 5.1 Three different joint types in ODE

A joint group is a special container that holds joints in a world. Joints can be added to a group, and then when those joints are no longer needed the entire group of joints can be very quickly destroyed with one function call. However, individual joints in a group can not be destroyed before the entire group is emptied. This is most useful with contact joints, which are added and remove from the world in groups every time step.

Table 5.1 shows some functions that deal with these objects, these functions take and/or return object IDs. The object ID types are dWorldID, dBodyID, etc.

Table 5.1: Some functions from ODE:

function	description
dWorldID dWorldCreate()	Create a new, empty world and return its ID number.
void dWorldDestroy (dWorldID)	Destroy a world and everything in it.
dSpaceID dHashSpaceCreate (dSpaceID space)	Create a multi-resolution hash table space. If space is nonzero, insert the new space into that space.
void dSpaceDestroy (dSpaceID)	destroys a space and all the geoms in that space.
dBodyID dBodyCreate (dWorldID)	Create a body in the given world with default mass parameters at position (0,0,0). Return its ID.
void dBodyDestroy (dBodyID)	Destroy a body. All joints that are attached to this body will be put into limbo.
dGeomID dCreateBox (dSpaceID space, dReal lx, dReal ly, dReal lz)	Create a box geom of the given x/y/z side lengths (lx,ly,lz), and return its ID. If space is nonzero, insert it into that space. The point of reference for a box is its center.
dJointID dJointCreateBall (dWorldID, dJointGroupID)	Create a new joint of a given type.
void dJointDestroy (dJointID)	Destroy a joint, disconnecting it from its attached bodies and removing it from the world.
dJointGroupID dJointGroupCreate (int max_size)	Create a joint group. The max_size argument is now unused and should be set to 0.
void dJointGroupDestroy (dJointGroupID)	Destroy a joint group. All joints in the joint group will be destroyed.
void dJointGroupEmpty (dJointGroupID)	Empty a joint group. All joints in the joint group will be destroyed, but the joint group itself will not be destroyed.

More details about the ODE's data types and functions will be mentioned in the following section.

5.2 Simulation environment development

5.2.1 The simulation process

A general flowchart for ODE simulation process is shown in Figure 5.2.

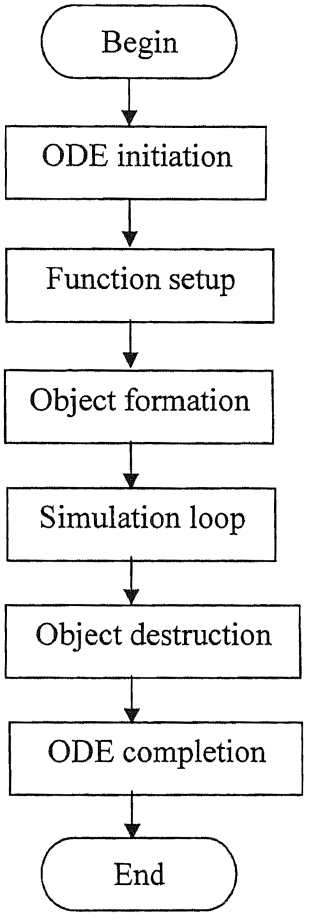


Figure 5.2 ODE simulation process

The "ODE initiation" process will call a function (dInitODE()) to initialize ODE library before first use. If this initialization succeeds the function may not be called again until ODE library is closed. Then the "function setup" process will initial

functions used in "simulation loop" process. The "object formation" process is to establish all the objects used in simulation. After these initiations, the "simulation loop" starts executing the simulation. When simulation is finish, all objects will be destroyed by "Object destruction" process. At last, the process "ODE completion" will call a function (dCloseODE()) to finish the simulation.

The backbone of ODE simulation is a function named as "dsSimulationLoop" and a structure named as "dsFunctions". Function "dsSimulationLoop" is used in "simulation loop" process to begin and complete a simulation process, it starts running a simulation, and only exits when the simulation is done. This function's parametric pointers should be provided for callbacks. The purpose of structure "dsFunctions" is to define several functions to be used as callbacks by the simulation loop; this structure will be initialled in the "function setup" process. Actually, "dsFunctions" is designed to provide callbacks for function "dsSimulationLoop". During simulation loop, these functions in structure dsFunctions will be called and executed one by one in accordingly order.

The definition of "dsSimulationLoop" and "dsFunctions" are as following:

```
void dsSimulationLoop (     int argc, char **argv,
                            int window_width, int window_height,
                            struct dsFunctions *fn
);

typedef struct dsFunctions {
```

```
    int version;                    /* put DS_VERSION here */
    void (*start)();                /* called before sim loop starts */
    void (*step) (int pause);       /* called before every frame */
    void (*command) (int cmd);      /* called if a command key is pressed */
    void (*stop)();                 /* called after sim loop exits */
    const char *path_to_textures;   /* if nonzero, path to texture files */
} dsFunctions;
```

As shown in the above definition, one of dsSimulationLoop's parameters is a dsFunctions structure, and some of structure dsFunctions's perameters are pointers to other functions. Using this method, function dsSimulationLoop can repeatedly use these functions defined by structure dsFunctions in a simulation loop.

In our program, the structure dsFunctions is initialled by the function setDrawStuff, which is defined as:

```
void setDrawStuff()
{
        fn.version = DS_VERSION;
        fn.start = &start;
        fn.step = &simLoop;
        fn.command = &command;
        fn.stop = 0;
        fn.path_to_textures = "../../drawstuff/textures";
}
```

After performing function "setDrawStuff", these functions start(), simloop() and command() will used in simulation loop, then the "simulation loop" process in figure 5.2 can be expanded as figure 5.3.

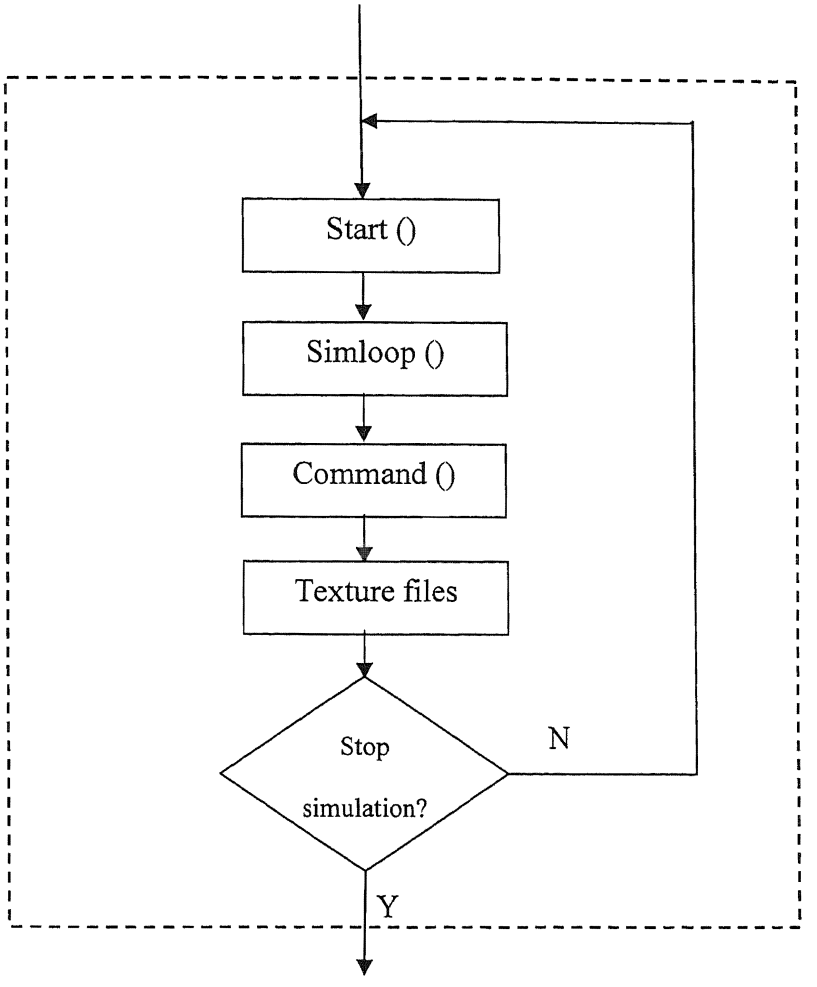


Figure 5.3 "Simulation loop" process

As mentioned before, the simulation is started and finished by process "simulation loop", and as shown in figure 5.3, there are only three functions in this process, that means, these three functions: start(), simloop() and command() will do the simulation work.

Function start() is to set the viewpoint of simulation and allocate thread local data allowing the thread calling ODE a simple. Function simloop() and command() are about controlling and displaying the service robot and its user, these two function will introduced in the following section.

5.2.2 Development of 3-D virtual world environment

The developed simulation environment is composed of a world and a space. This environment has a plane, which is used to imitate ground on the earth, to hold everything used in the simulation, such as rigid bodies with associated geometries and masses and joint groups to store the contact joints that are created during a collision. This environment has a collision detection system which is used to handle all the contacts between simulation objects. To mimic the real environment, the gravity of this simulation environment is exactly the same as the gravity on the earth.

The following functions create an empty world and an empty space for the simulation enrironment:

```
world   =   dWorldCreate();
space   =   dHashSpaceCreate(0);
```

The established world's default global gravity vector is (0, 0, 0). The following function changes the world's global gravity vector to earth's gravity vector:

```
dWorldSetGravity(world, 0, 0, -9.8);
```

The ground of the simulation environment is defined by Equation (5.1):

Z=0 (5.1)

To implement this ground, the following function is used:

dGeomID dCreatePlane (dSpaceID space, dReal a, dReal b, dReal c, dReal d);

The function "dCreatePlane" can create a plane geometry according to the given parameters, the created plane's equation is:

a*x+b*y+c*z = d (5.2)

If a, b, c, d in Equation(5.2) is set to 0,0,1,0 correspondingly, Equation(5.2) will become Equation(5.1), then the ground can be gotten.

In our program the function dCreatePlane's is :

ground = dCreatePlane(space, 0, 0, 1, 0);

This creates the ground and inserts this ground into simulation environment.

Until this stage, the world and space with gravity and ground are all created, to enable them work together to simulate a 3-D virtual reality environment, the collision detection engine mentioned in Section 5.1 needs to be implemented. Two functions are used for this purpose:

void dSpaceCollide (dSpaceID space, void *data, dNearCallback *callback);

int dCollide (dGeomID o1, dGeomID o2, int flags, dContactGeom *contact, int skip);

In our program the function dSpaceCollide is:

dSpaceCollide(space,0,&nearCallback);

This function will call function "nearCallback" to get contact points between two geometries. This function is the first function in function "simloop". simloop is composed of several functions, these function will be executed one by one to perform simulation. Figure 5.4 is the flowchart of function simloop:

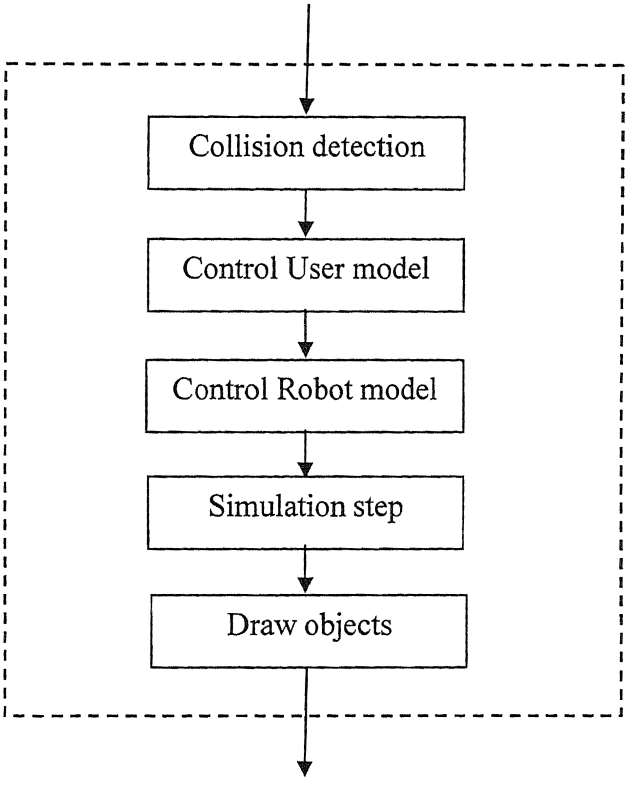


Figure 5.4 Function "simloop"

"Collision detection" will change all objects dynamical parameters accordingly. "Control user model" and "control robot model" are interfaces to control the simulated human user and service robot; "simulation step" executes one step simulating; and "draw object" will update and display the current simulation result.

Figure 5.5 shows developed the world only with a red ball on ground.

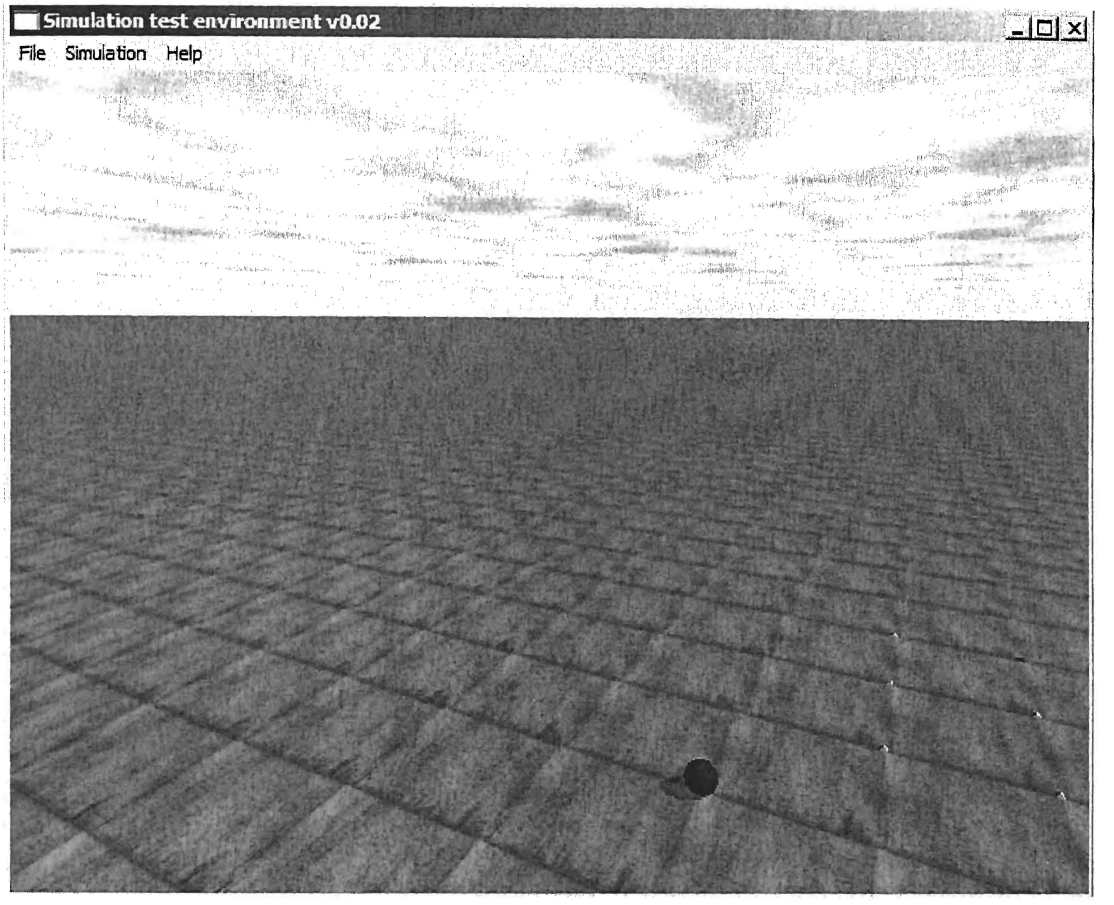


Figure 5.5 The 3-D virtual world environment

5.2.3 Development of a service robot and its user model

To simulate a service robot working together with its user, a robot model and a human user model are developed in the proposed simulation environment. These models should have the capability to manipulate an object, such as holding and moving a box, to cooperate with each other.

The developed service robot model and its user model are shown in Figure 5.6.

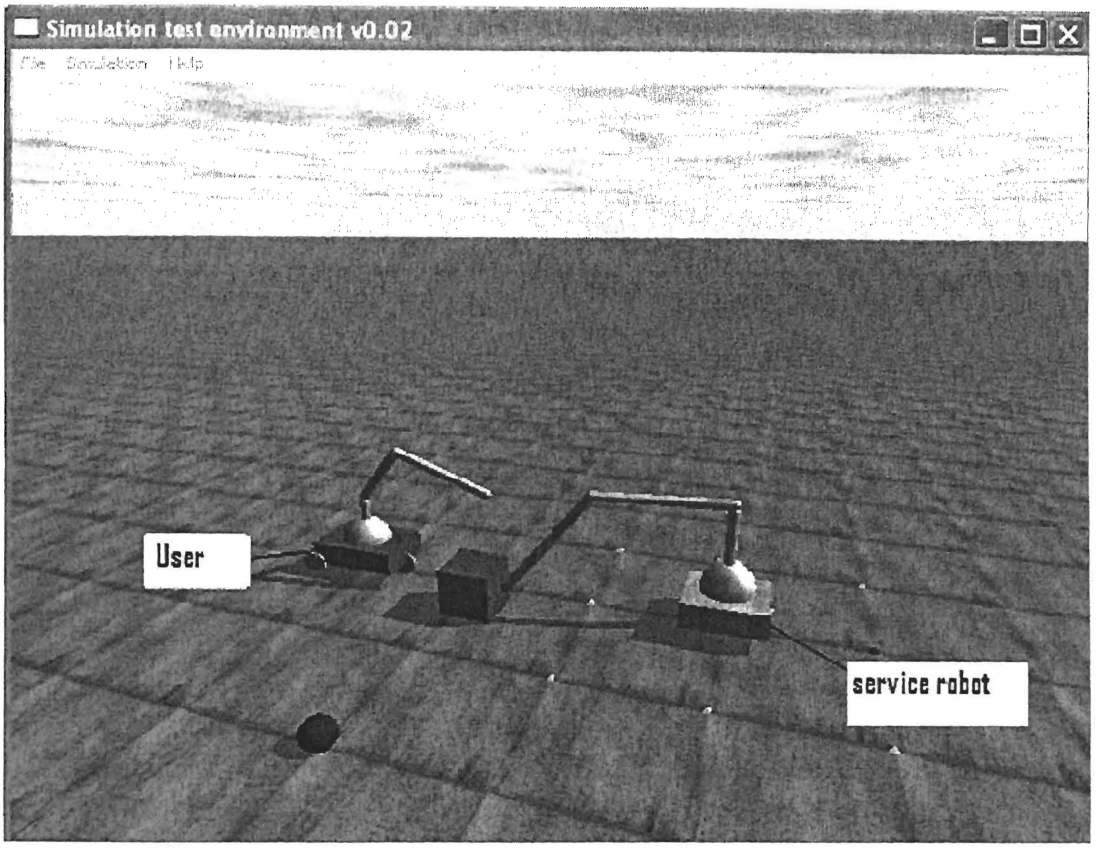
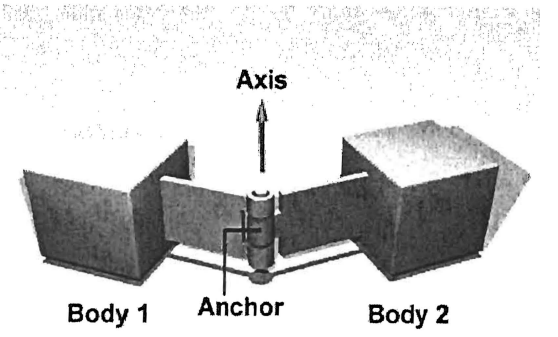


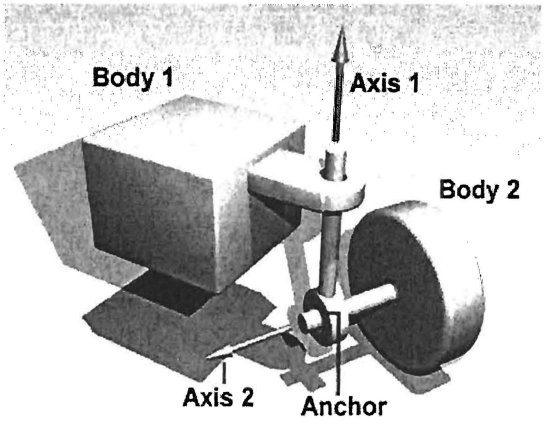
Figure 5.6 Models of service robot and its user

Both the service robot and its user are composed of a robot arm and a movable base. The difference between them is the service robot is controlled by a special function while the user is controlled by keyboard.

Hinge joints, shown in Figure 5.7(a), are used to establish the robot arm; hinge-2 joint, shown in Figure 5.7(b), are used to create the movable base.



(a) A hinge joint



(b) A hinge-2 joint

Figure 5.7 Two hinge used in robot model

These two kinds joint will connect cylinders and boxes together to establish the robot and user model. Figure 5.8 displays the structure details of the robot model.

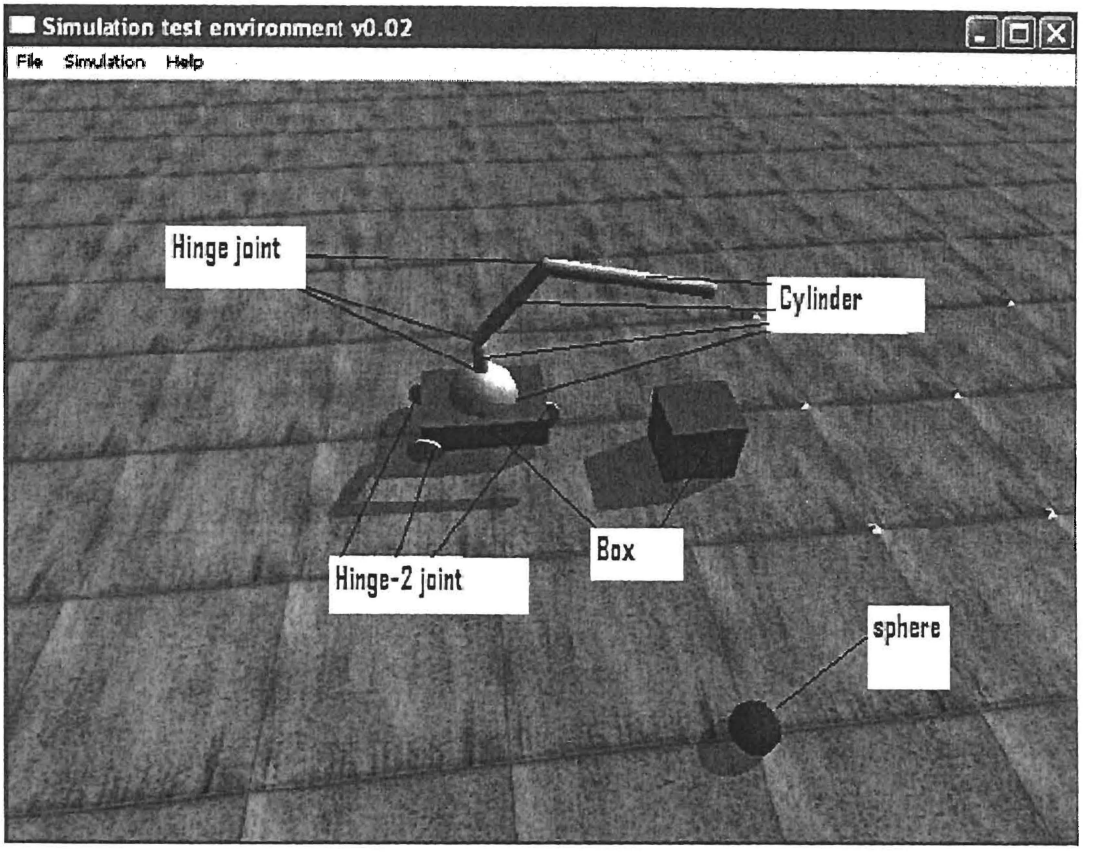


Figure 5.8 Structure details of the robot model

The human user model is controlled by keyboard, the used buttons and their functions are introduced in Table 5.2

If the keyboard is used, the "command" function mentioned in Figure 5.3 will capture the active button and corresponding program to be performed to control the user.

The robot is controlled by a function; this function can control the robot's actions according the user's actions in a moving box task process. Figure 5.9 shows the service robot and its user lift the box together.

Table 5.2: The buttons used for user controlling and their functions

button	function
W(w)	to increase speed
S (s)	to decrease speed
A (a)	to steer left
D (d)	to steer right
SPACE	to reset speed and steering
I (i)	to lift the upper arm up
K (k)	to drop the upper arm down
Q (q)	to lift the forearm up
Z (z)	to drop the upper arm down
J (j)	to turn the arm left
L (l)	to turn the arm right

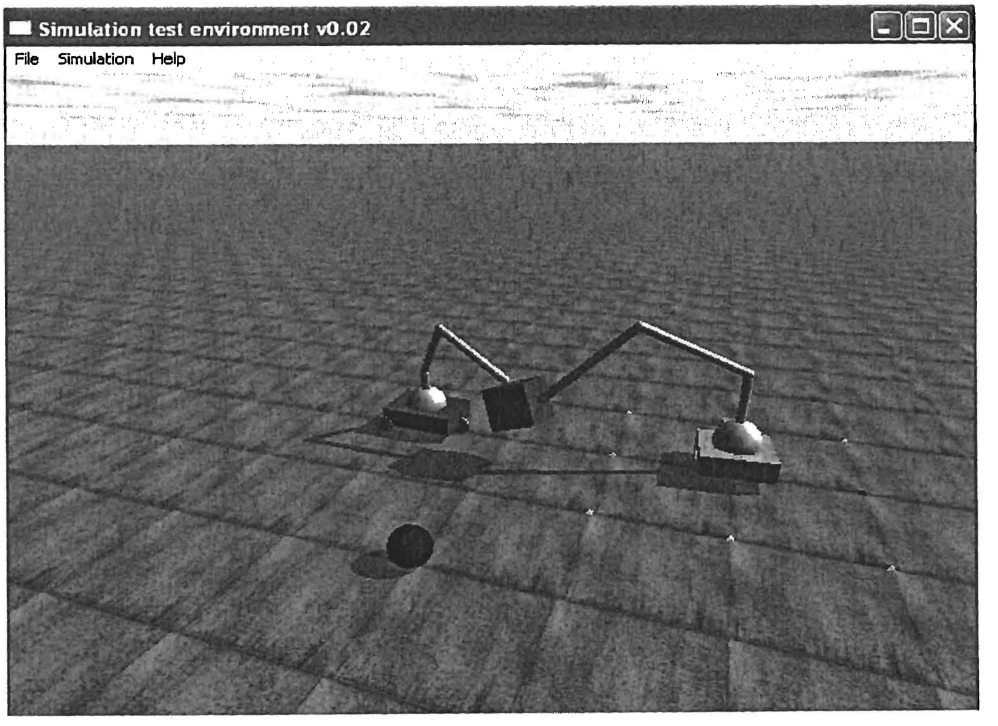


Figure 5.9 The robot and its user work together

Section 5.4 will give more details about the robot and its user work together and will show how the robot performs test actions.

5.3 Integration

5.3.1 Task classifier and TAB integration

A task classifier based on verb's feature vector was established in Chapter 3, the test results shown in section 3.3.4 demonstrated this task classifier can successfully extract tasks from user's commands and classify them into task categories; the proposed TAB was developed in Chapter 4 with test actions and taught tasks in a hierarchical structure from the general to the specific. In this section the task classifier, a C++ program, and the TAB, implemented using MySQL, will be integrated together. After this integration the TAB can provide test actions according user's commands.

The "MySQL Connector/C++" is a MySQL database connector driver for C++, this driver can be used to connect to MySQL 5.1 or later version. To use "MySQL Connector/C++" in our program, one head file "mysql_public_iface.h" needs to be included, the MySQL Connector/C++ static library file "mysqlcppconn-static.lib" needs to be statically linked, and the following two files from MySQL database is needed as well: libmysql.dll and libmysql.lib. After including these files, our program becomes able to access MySQL database, that is, the task classifier can work with TAB together.

Figure 5.10 shows the outcome of integrated task classifier and TAB. As displayed, our program can using the task category from a task classifier to retrieve and display all the taught tasks belonging to that task category from TAB, the specific task according to a user's command in that category can be located, and its corresponding

test actions can be picked out. For more information about the test actions retrieval process, see the Section 4.3.

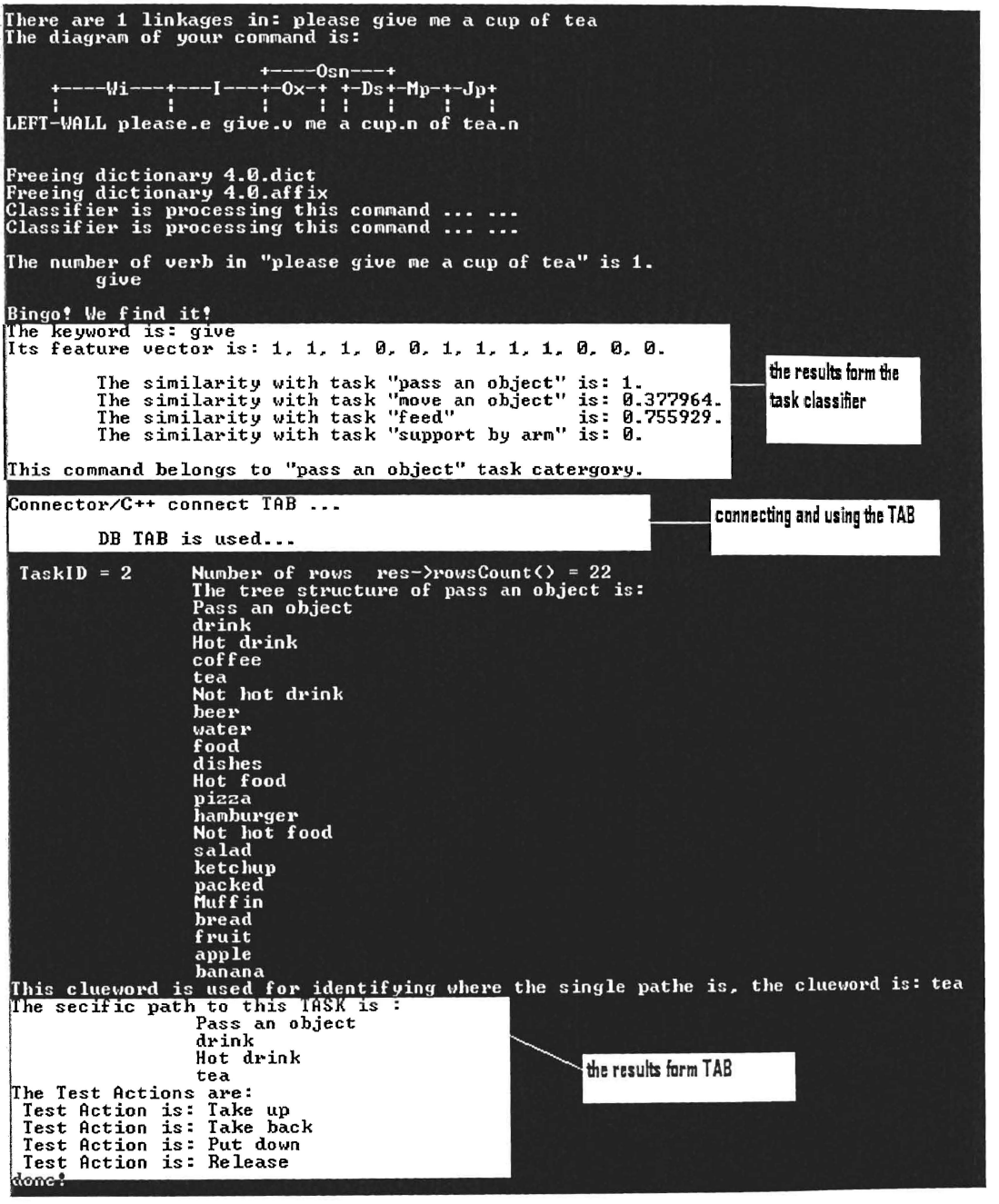


Figure 5.10 The result from integration of task classifier and TAB

5.3.2 Task classifier and TAB in simulation environment

In Section 5.2 a robot model and a human user model were developed in the proposed simulation environment in order to simulate a service robot working together with its user. This section explains how these models can be used to demonstrate the performance of test actions during a cooperative task according to user's command. As these models are able to manipulate an object, the task of holding and moving a box is used in this simulation.

In our simulation, to enable the task classifier to work, the user needs to use button "c". After pressing button "c", the task classifier is activated and it will ask the user to input his commands, as shown in Figure 5.11.

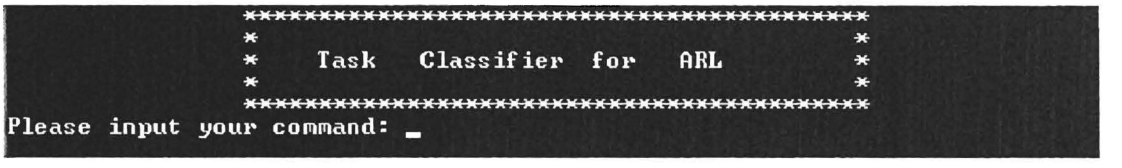


Figure 5.11 Task classifier interface in simulation environment

After having received a command, the task classifier will try to find its corresponding category, then to pick up test actions belonging to this category from TAB, as shown in Figure 5.9. If the user model stays still or stops moving for more than three seconds during the cooperative process, the robot model will perform test actions. Figure 5.12 shows the robot performing the test action of "move up". Figure 5.13 displays the cooperative task, all test actions belonging to this task category, and the test action the robot performed.

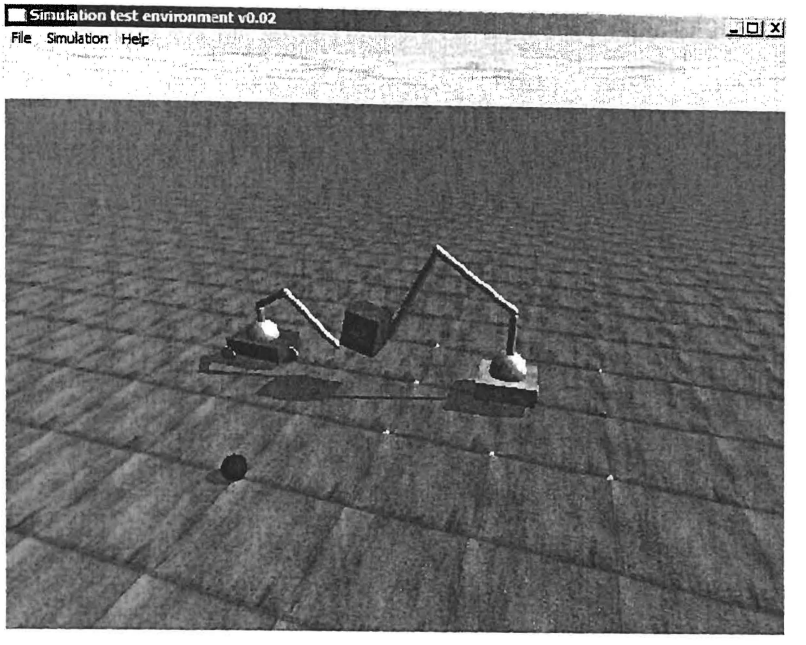


Figure 5.12 Performance of the test action "move up"

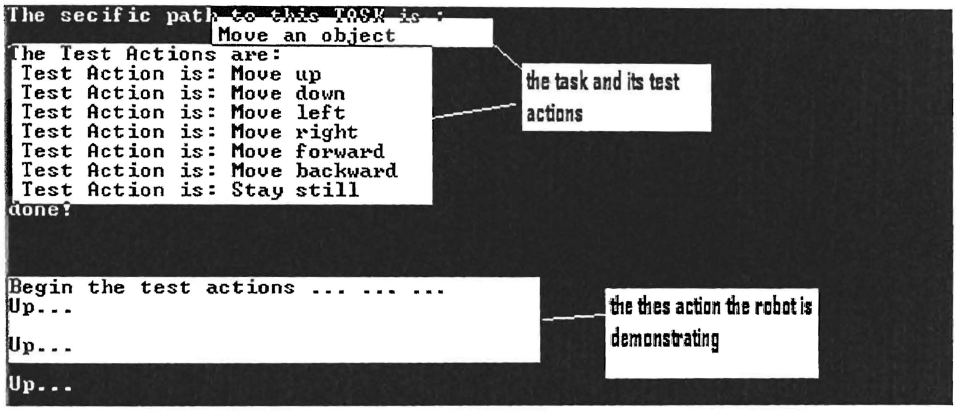


Figure 5.13 All test actions and the one performed

5.4 Simulation result

5.4.1 Provision of test actions by TAB

The operation of the integrated task classifier and TAB for three more tasks "bring me a glass of water", "give me a pint of beer", and "give me an apple", belonging to the category of "pass an object", are shown in Figures 5.14 to 5.16.

Figure 5.14 Test actions for task "please bring me a glass of water"

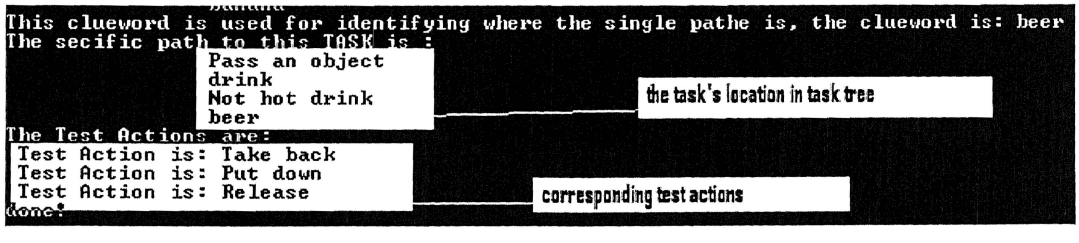


Figure 5.15 Test actions for task "could you give me a pint of beer"

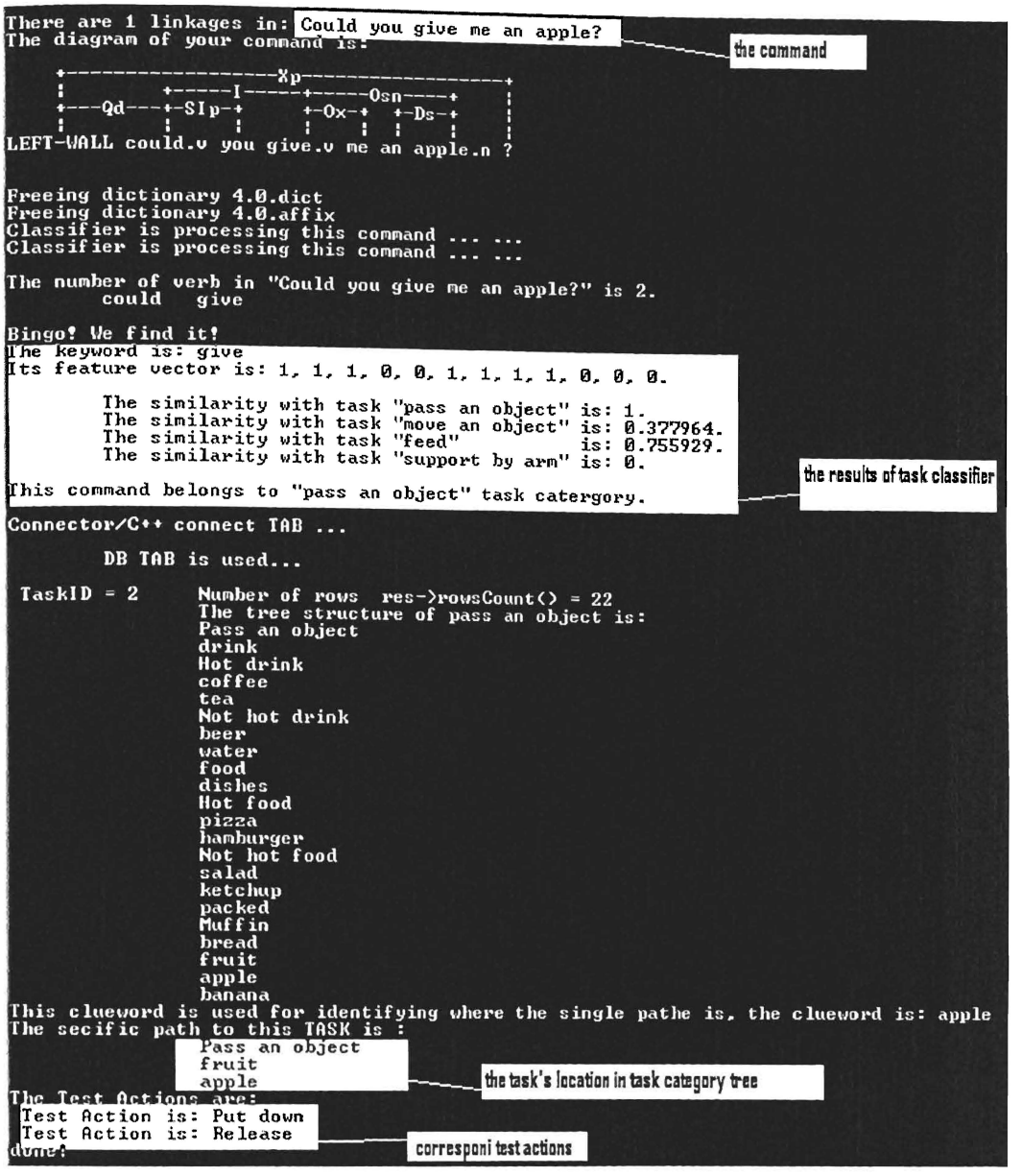


Figure 5.16 Test actions for task "please give me an apple"

For the remaining three tasks category, two simulation results of each category are given as follows.

Figure 5.17 shows the test actions for task "help me walk to the upstairs". Figure 5.18 shows the test actions for task "please help me walk downstairs". Figure 5.19 shows the test action for task "could you help me walk forward".

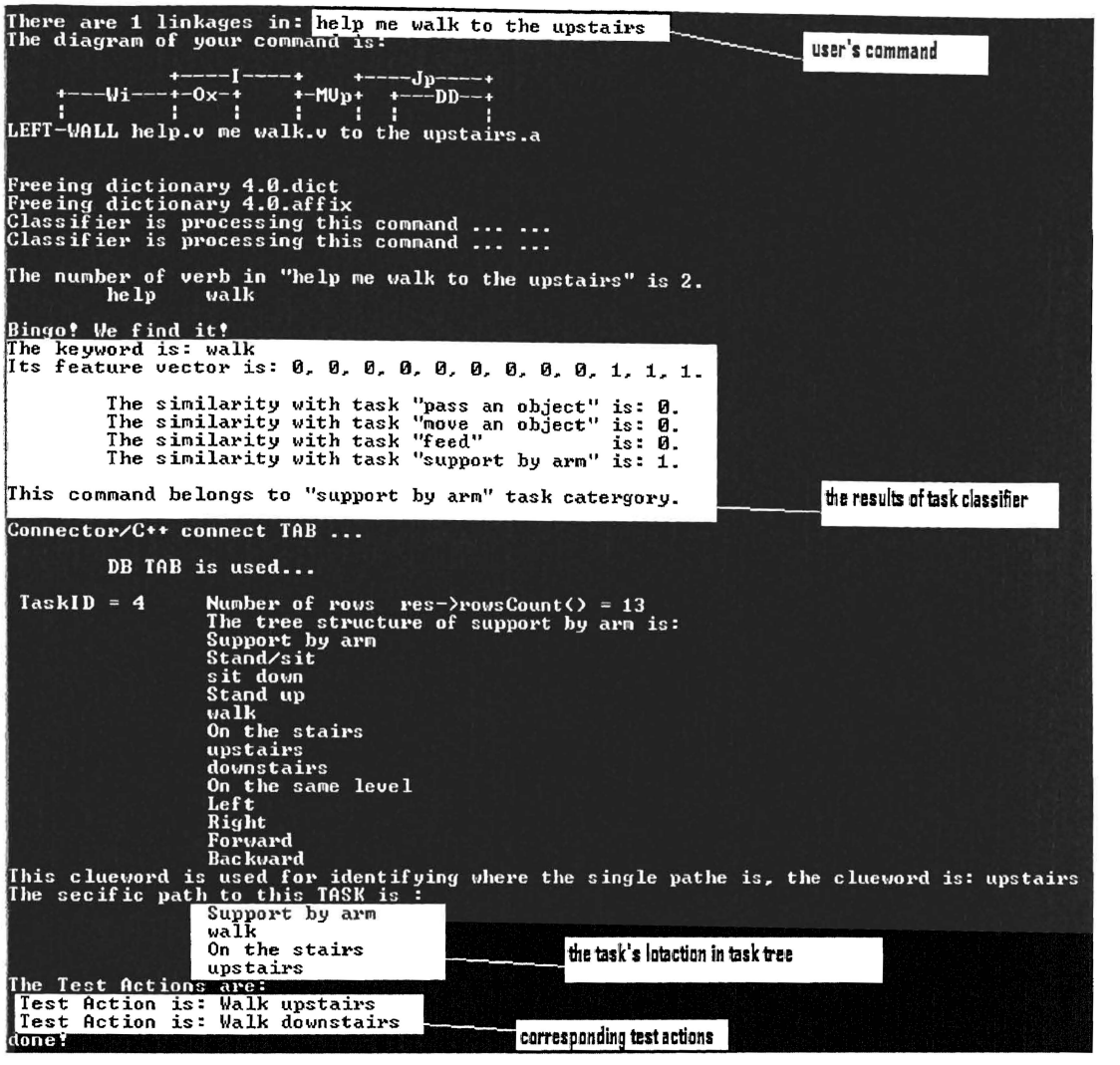


Figure 5.17 Test actions for task "help me walk to the upstairs"

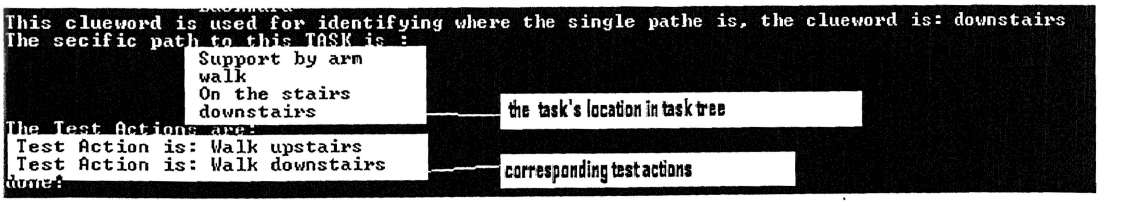


Figure 5.18 Test actions for task "please help me walk downstairs"

Figure 5.19 Test actions for task "could help me walk forward"

Figure 5.20 shows the test actions for task "please help me move this bed". Figure 5.21 shows the test action for task "help me carry this bucket of water". As mentioned in Section 5.3, test actions of this category can be demonstrated by the service robot model, these demonstrations will be displayed in section next Section 5.4.2.

Figure 5.20 Test actions for task "please help me move this bed"

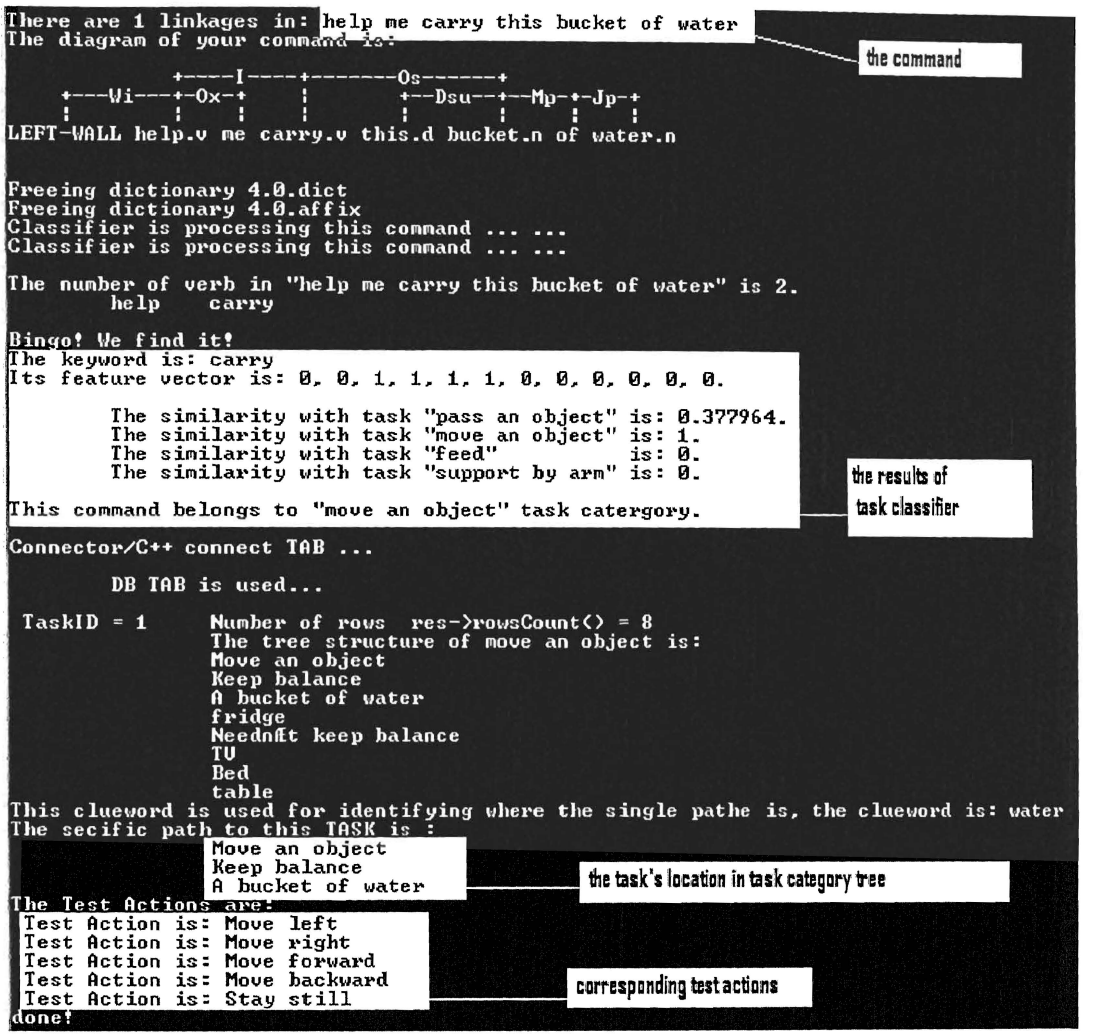


Figure 5.21 Test actions for task "help me carry this bucket of water"

Figure 5.22 shows the test actions for task "please feed some pizza to me". Figure 5.23 shows the test action for task "could you help me drink my soup".

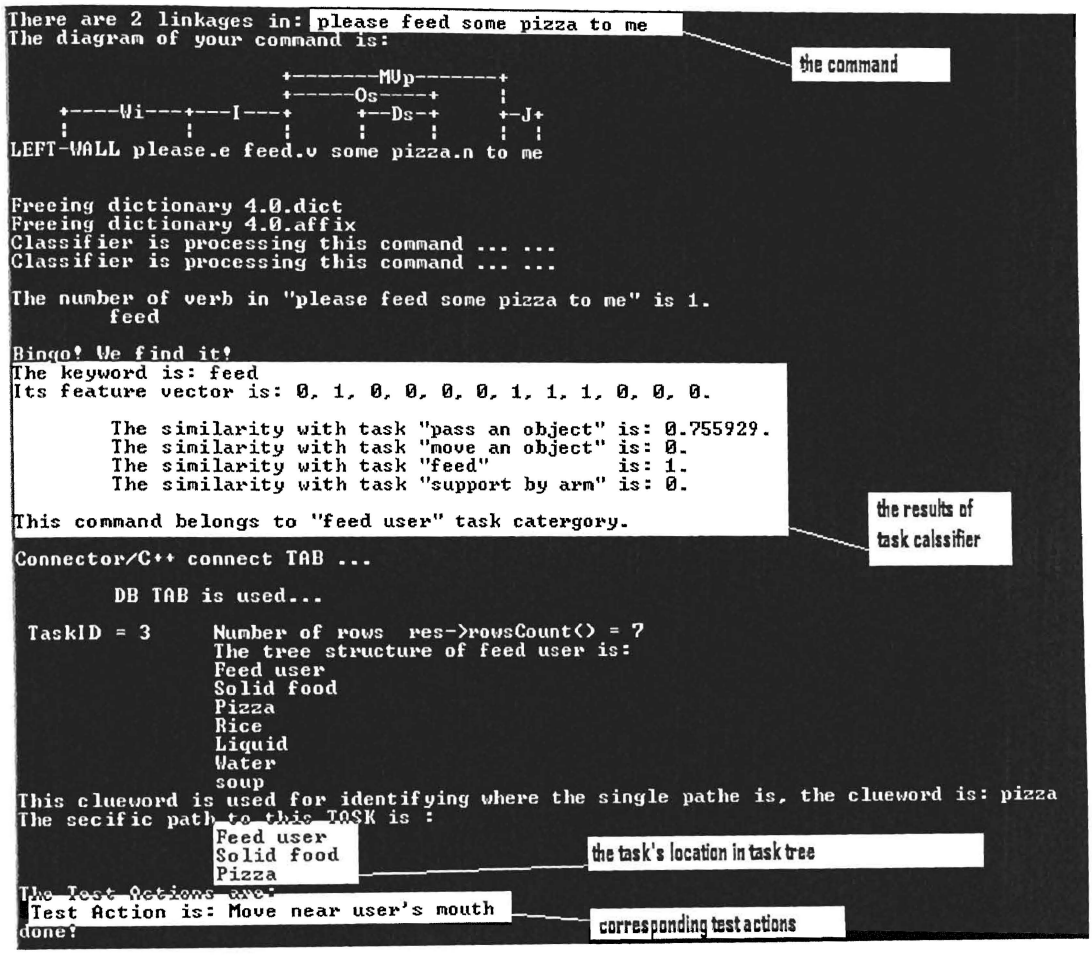


Figure 5.22 Test actions for task "please feed some pizza to me"

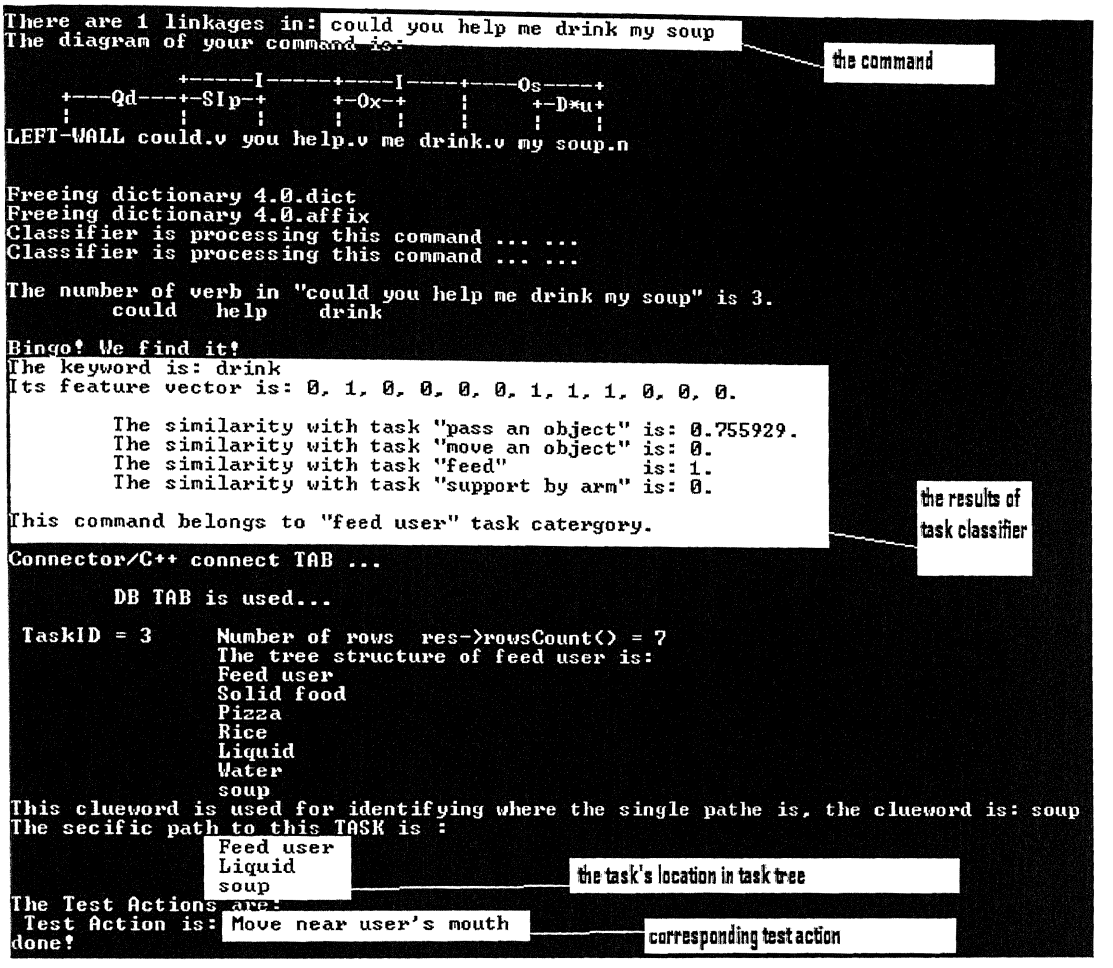


Figure 5.23 Test actions for task "could you help me drink my soup"

From the results shown above, the TAB shows its capability to provide test action for different task, more discussion about these test actions will be given in Section 5.5.

5.4.2 Performance of test actions

This section will demonstrate the remaining test actions of "help user to move an object" category mentioned in Figure 5.11. Figure 5.24 to 5.28 show the test action of "move down" "move left", "move right", "move forward" and "move backward,

respectively. Figure 5.29 shows the corresponding test actions the robot performed in simulations.

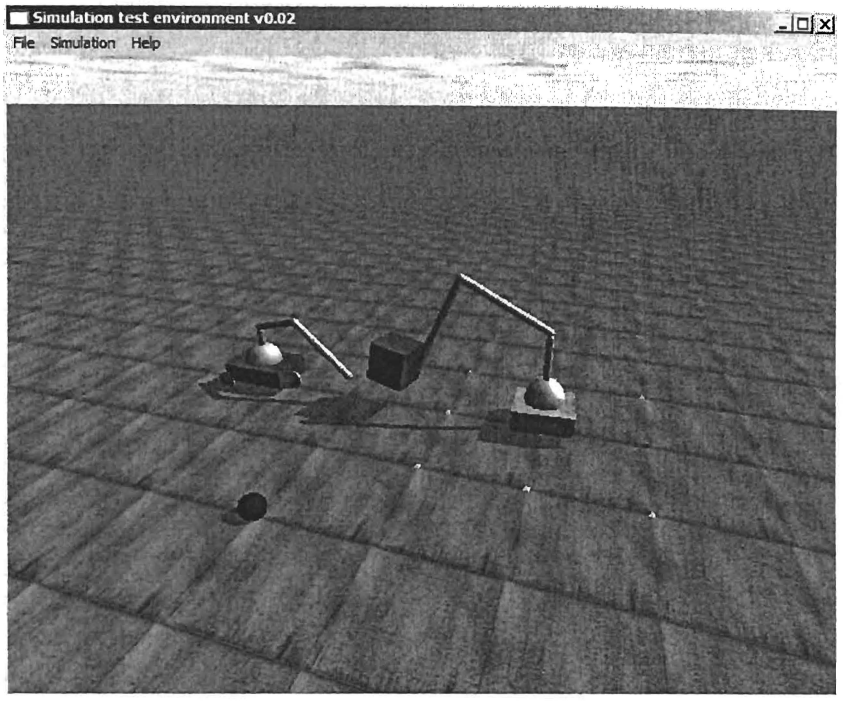


Figure 5.24 Robot performs "move down" test action

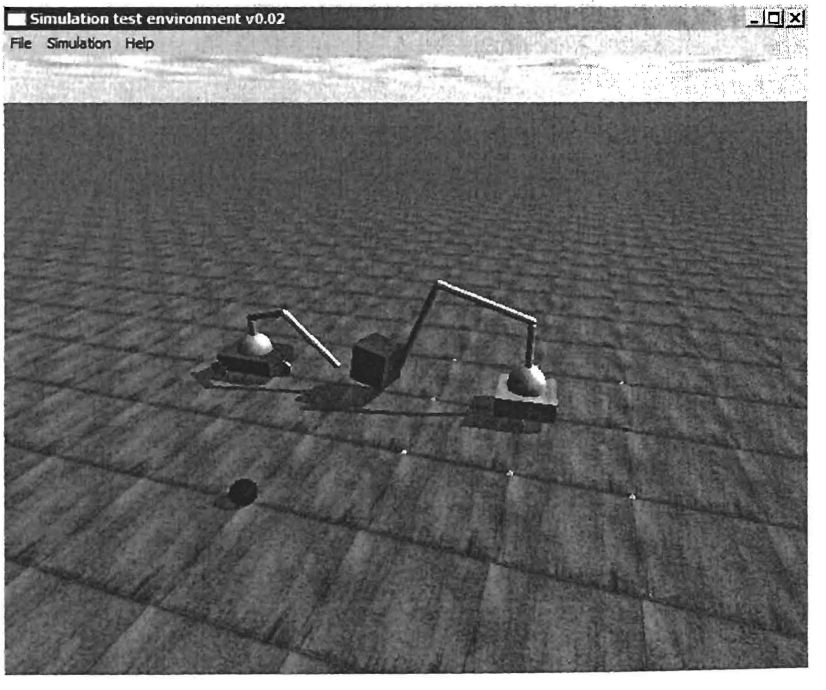


Figure 5.25 Robot performs "move left" test action

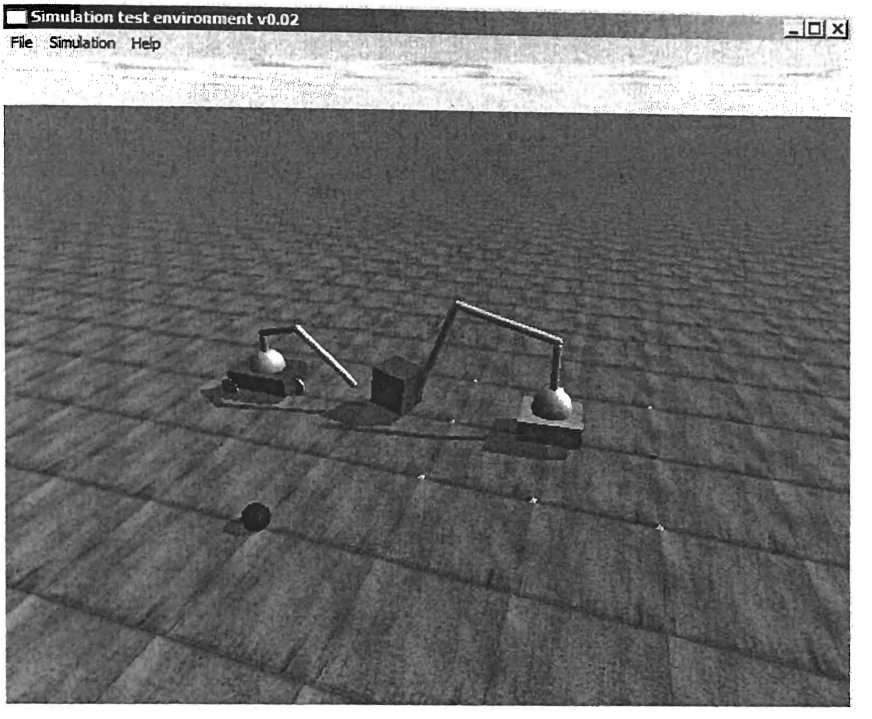


Figure 5.26 Robot performs "move right" test action

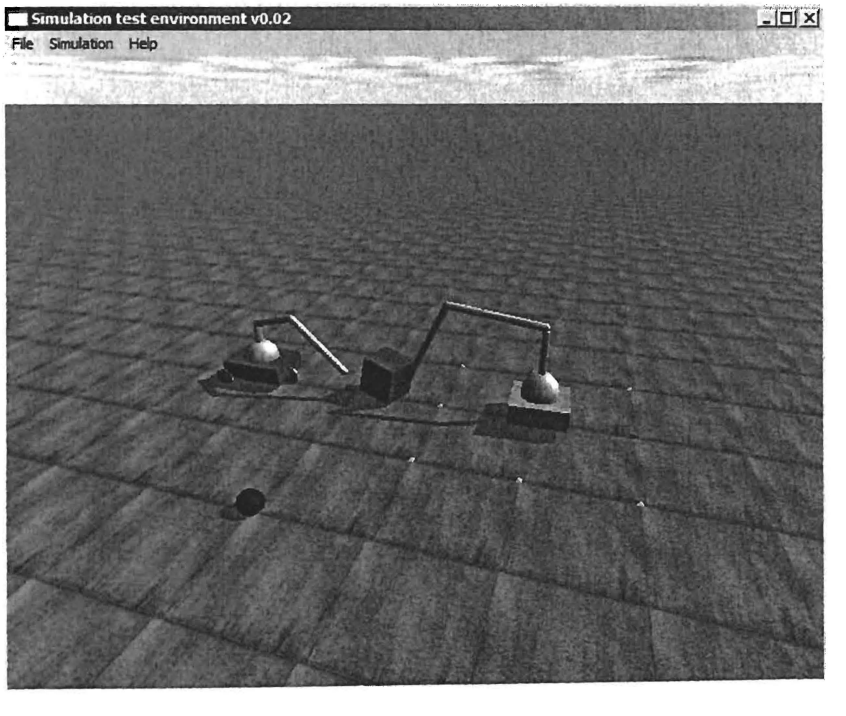


Figure 5.27 Robot performs "move forward" test action

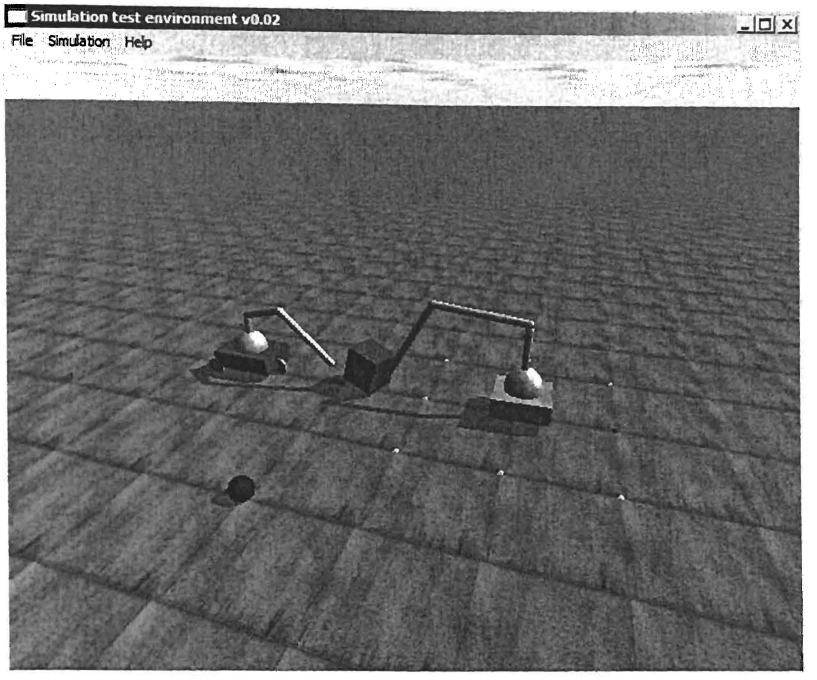


Figure 5.28 Robot performs "move backward" test action

```
Up...
Up...
Up...
Down...
Down...
Down...
Left...
Left...
Left...
Left...
Right...
Right...
Right...
Forward...
Forward...
Forward...
Backward...
Backward...
Backward...
The end of Test actions simulation ... ...
The end of Test actions simulation ... ...
The end of Test actions simulation ... ...
```

Figure 5.29 Test actions the robot performed

5.5 Discussion

In Section 5.4.1, from Figure 5.12 to Figure 5.21, ten selected commands were chosen to represent four proposed typical task categories to test the TAB. As shown in these diagrams, the TAB successfully provides corresponding test actions for the different tasks. In Section 5.4.2, the proposed service robot model performed six test actions of "help user to move an object" task category. These demonstrations proved these test actions can be used in a cooperative task.

As the results demonstrated, different tasks will have different test actions. However, some tasks share the same test actions, such as task "please bring me a glass of water" and task "could you give me a pint beer. These two tasks belong to task "pass not hot drink". Task "help me walk to the upstairs" and "please help me walk downstairs" are in the similar situation, that is, they belong to task "support user walk on the stairs", so they share the same test actions.

CHAPTER 6. CONCLUSIONS AND FURTHER WORK

6.1 Conclusions

This study aimed to develop a test action bank (TAB) for ARL. The proposed TAB is composed of test actions and taught tasks, which are represented in hierarchical structure from the general to the specific. This study also developed a verb-based task classifier which is used to extract tasks from user's commands and to classify them into task categories. At the end, this study sets up a simulation environment to simulate and test the applicability of the task classifier and the TAB.

At the beginning of this study, a survey of the state-of-the-art approaches in the area of interaction between robot and humans shows that ARL is a promising approach. The distinct advantage of ARL is its active learning approach. TAB is one critical component for implementing ARL, it can provide test actions used by a robot during cooperative tasks. To develop TAB, the following work has been done:

- Generalizing four typical task categories, from which several specific taught tasks and their corresponding test actions are derived. These four typical task categories are: "pass an object", "support by arm", "feed the user" and "help user to move an object".
- Defining verb's feature by relevant nouns and using these noun verbs establishing verbs feature vectors, which can be used to calculate distance between verbs. These defined verbs are used to map commands to task categories.

- Establishing a verb-based task classifier to abstract task from user's command, this task classifier is used to classifier users' commands into categories at an abstract level, this is helpful to locate a task and pick out its corresponding test actions from TAB.
- Storing task trees into relational database using a hierarchical structure with the top are very abstract tasks and the very specific tasks are arranged at the bottom to enable a group of similar tasks to share sets of test actions.
- Establishing TAB in MySQL database and using a backward retrieval method to retrieve a specific task's corresponding test action from TAB.
- Developing a 3-D VR environment with a service robot and a user model to test the TAB. This simulation environment can find out corresponding test actions according to user's command and display these test actions.

All the experiments and tests results in this dissertation show that:

- The proposed task classifier can successfully abstract task from user's commands.
- The developed TAB is able to provide corresponding test actions according to different tasks.

6.2 Further Work

More typical task categories can be generalized to derive more taught tasks and corresponding test actions used in establishing TAB.

Improving the method used to define a verb's feature by relevant nouns, statistical method can be added as supplements. Mutual information of two variables is a quantity that measures the mutual dependence of the two variables (Church and Hanks, 1989). Mutual information statistics, therefore, can be used to identify co-occurrence between verbs and nouns given large corpora. Comparing the probability of observing verb and noun together (the joint probability P(*verb*, *noun*)) with the probabilities of observing verb and noun independently(the chance probability P(*verb*) and P(*noun*)). If there is a genuine association between verb and noun, then the joint probability P(*verb*, *noun*) will be much larger than chance P(*verb*)*P(*noun*). If there is no interesting relationship between verb and noun, then P(*verb*, *noun*) will be almost the same as P(*verb*)*P(*noun*). If verb and noun are in complementary distribution, then P(*verb*, *noun*) will be much less than P(*verb*)*P(*noun*). Probability P(*verb*) and P(*noun*)), are estimated by counting the number of observations of verb and noun in a corpus, and normalizing by N, the size of the corpus. Joint probabilities, P(*verb*, *noun*), are estimated by counting the number of times that verb is followed by noun, and normalizing by N.

Employing further natural language processing techniques in this study make the task classifier more robust and capable of processing more variety of commands, furthermore, to make the test action retrieval method more precise.

Improving and expanding the structure of TAB to be more scientific and reliable. More tables can be added into TAB and more attributes are needed to make task classify and test actions retrieve more precise.

REFERENCES

SRI Consulting Business Intelligence (SRIC-BI), Disruptive Civil Technologies Six Technologies with Potential Impacts on US Interests out to 2025,APPENDIX E: SERVICE ROBOTICS (BACKGROUND), April 2008.

C. Breazeal, A. Brooks, D. Chilongo, J. Gray, G. Hoffman, C. Kidd, H. Lee, J. Lieberman, A. Lockerd, "Working Collaboratively with Humanoid Robots", Proceedings of the IEEE/RAS Fourth International Conference on Humanoid Robots (Humanoids 2004), Los Angeles, CA. 253-272, 2004.

C. Breazeal, A. Brooks, J. Gray, G. Hoffman, C. Kidd, H. Lee, J. Lieberman, A. Lockerd, and D. Chilongo, "Tutelage and Collaboration for Humanoid Robots," International Journal of Humanoid Robots,1(2), 315—348, 2004.

Edsinger, Aaron and Kemp, Charles, "Human-Robot Interaction for Cooperative Manipulation: Handing Objects to One Another", Proceedings of the IEEE International Workshop on Robot and Human Interactive Communication (ROMAN), 2007.

Oliver C. Schrempf, Uwe D. Hanebeck, Andreas J. Schmid., Heinz Wörn, A Novel Approach to Proactive Human-Robot Cooperation, Proceedings of the 2005 IEEE International Workshop on Robot and Human Interactive Communication (ROMAN 2005), pp. 555-560, Nashville, Tennessee, August, 2005.

Oliver C. Schrempf, Uwe D. Hanebeck, A Generic Model for Estimating User Intentions in Human-Robot Cooperation, Proceedings of the 2nd International Conference on Informatics in Control, Automation and Robotics (ICINCO 2005), 3:251-256, Barcelona, Spain, September, 2005.

Oliver C. Schrempf, Andreas J. Schmid, Uwe D. Hanebeck, Heinz Wörn, Towards Intuitive Human-Robot Cooperation, 2nd International Workshop on Human Centered Robotic Systems (HCRS 2006), pp. 7-12, Munich, Germany, October, 2006

Tapus, A. and Mtaraic, M.: Towards active learning for socially assistive robots, Robotics Challenges for Machine Learning Workshop, the 21st Annual Conference on Natural Information Proceeding Systems, December, Vancouver, B.C., Canada, 2007.

Li D., Liu B., Maple C., Jiang D. and Yue Y.: Active robot learning for building up high-order beliefs, Proceedings of the 5th International Conference on Fuzzy Systems and Knowledge Discovery, Jinan, China, August, 2008

Calinon, S. and Billard, A.: Teaching a humanoid robot to recognize and reproduce social cues, The 15th IEEE International Symposium on Robot and Human Interactive Communication, Hatfield, UK, pp. 346-351, September, 2006.

M. Anderson, A. Oreback, and H. I. Christensen,"ISR: An intelligent service robot," in Intelligent Sensor Based Robotics (H. I. Christensen,H. Bunke, and H. Noltemeier, eds.), Springer Verlag, November 1999.

Hyoung-Rock Kim, Dong-Soo Kwon, Task Modeling for Intelligent Service Robot using Hierarchical Task Analysis, Proc. of the 2004 FIRA Robot World Congress, Busan, Korea, 26-29 October, 2004

Shepherd, "Hierarchical Task Analysis", Talyor & Francis, 2001

H. Eftring, the Useworthiness of Robots for People with Physical Disabilities, PhD thesis, CERTEC, Lund University, Lund, Sweden, 1999.
http://www.certec.lth.se/doc/useworthiness/ as of 2000/07/01.

A. Green, H. Hüttenrauch, L. Oestreicher, K. Severinson-Eklundh, and M. Norman "User Centered Design for Intelligent Service Robots", in Proc. of Ro-Man 2000, Osaka. Japan, September 2000.

D. Jurafsky,and J. Martin, Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics, 2000a, pp.285-318.

D. Jurafsky,and J. Martin, Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics, 2000b, pp.285-318.

Guo X., Li D., Clapworthy G. and Pritchett N., "A conditional mutual information based selectional association and word sense disambiguation," in proceedings of IEEE NLP-KE 2009, pp. 249-255, 2009.

Foltz, P. W., Kintsch, W.,& Landauer, T. K, The measurement of textual Coherence with Latent Semantic Analysis. Discourse Processes, 25, 285-307, 1998.

Foltz, P. W., Latent Semantic Analysis for text-based research. Behaviour Research Methods, Instruments and Computers, 28, 197-202, 1996.

S. Deerwester, Susan Dumais, G. W. Furnas, T. K. Landauer, R. Harshman, "Indexing by Latent Semantic Analysis" . Journal of the American Society for Information Science 41 (6): 391–407, 1990.

Daniel Sleator and Davy Temperley, Parsing English with a Link Grammar, Carnegie Mellon University Computer Science technical report CMU-CS-91-196, October 1991

Link Grammar Documentation: The Link Parser Application Program Interface (API), http://www.link.cs.cmu.edu/link/api/index.html

Chris Newman, Sams Teach Yourself MySQL in 10 Minutes, Published May 9, 2006 by Sams.

E.F Codd, "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM 13 (6): 377–387, 1970.

Gijs Van Tulder, Storing Hierarchical Data in a Database, 2003
http://articles.sitepoint.com/article/hierarchical-data-database/1

Joe Celko, Trees and Hierarchies in SQL for Smarties, excerpt from Chapter 2: "Adjacency List Model". ISBN 1-55860-920-2, Morgan Kaufmann, 2004.

Peter Pin-shan Chen, The Entity-Relationship Model: Toward a Unified View of Data, ACM Transactions on Database Systems, pages9-36, 1976

Open Dynamics Engine (ODE) Community Wiki:
http://opende.sourceforge.net/wiki/index.php/Main_Page

ODE's official website: http://www.ode.org

Russell Smith, Open Dynamics Engine v0.5 User Guide, 2006,
http://www.ode.org/ode-latest-userguide.html

Kosei Demura, Robot Simulation - Robot Programming with Open Dynamics Engine, ISBN:978-4627846913, Morikita Publishing Co. Ltd., Tokyo, 2007
http://demura.net/simulation

MySQL 5.1 Reference Manual, http://dev.mysql.com/doc/refman/5.1/en/index.html

Church K, and Hanks P: Word Association Norms, Mutual Information, and Lexicography, ACL Proceedings, Also (to appear) in Computational Linguistics, 1989